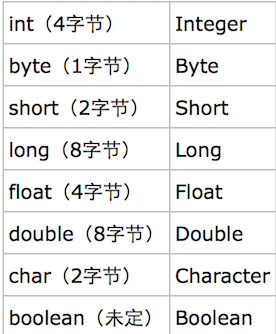
1. **装箱拆箱**，**继承多态**（方法重载和方法重写）**this关键字**（总是指向该方法调用的对象）

装箱就是将自动类型转换为包装类型，拆箱就是自动将包装类型转换为基本数据类型

装箱的过程会创建对应的对象，这个会消耗内存，所以装箱的过程会增加内存的消耗，影响性能。



基本类型-》包装类型：可以通过new 或者相应包装类的valueof（）方法。

包装类型-》基本类型：可以通过Xxxvalue()得到

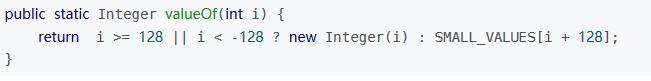
例如：Integer i1 = new Integer(6)/integer i2 = Integer.valueof(6)(会缓存）

Int i3 = i2.intvalue().

基本类型-》string:string.valueof()

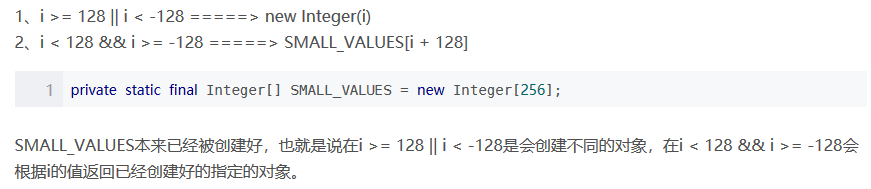
Sring-》基本类型： int n = Integer.parseInt(string);

**对于int型**数据而言，他可以通过integer.valueof(6),生成一个包装类型，其中Integer的valueof源码：





它会首先判断i的大小，如果-128<i<128,则通过SMALL\_VALUES[i + 128]，它是一个静态的integer数组对象，也就是valueof返回的都是integer对象，否则就创建一个Integer对象



对于Double，它自动装箱每次都会生成新的对象。

Integer派类：Integer,Short,Byte,Charactor,long这几个类的valueof方法实现是类似的

Double派别：Double,float的valueof方法的实现是类似的，每次返回不同的对象

而对于Booolean的valueOf返回的都是相同的对象。

对于一个基础数据类型与封装类进行==，+，-，\*，/运算时，会将封装类进行拆箱，对基础类型进行运算，但必须满足类型相同和内容相同，当 “==”运算符的两个操作数都是 包装器类型的引用，则是比较指向的是否是同一个对象，而如果其中有一个操作数是表达式（即包含算术运算）则比较的是数值（即会触发自动拆箱的过程）

**多态**就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量到底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。**因为在程序运行时才确定具体的类，这样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上**，从而导致该引用调用的具体方法随之改变，让程序可以选择多个运行状态动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

实例变量不具备多态性，

编译时多态（这种多态在编译时已经确定好了）：

方法重载和方法重写，一种是静态分派（发生在编译器），一种是动态分派（发生在运行期），而对于多态他也相应的分为静态类型和实际类型，静态类型在编译是就可以确定，但实际类型必须等到程序运行时才能够确定下来，在静态分派时，使用哪个重载版本，就完全取决于传入的参数类型和数量，如果有一个父类，他有两个子类，三者都实现了同一个方法，并且方法的参数是自己的实例对象，在进行多态调用时，编译器在重载是是通过静态类型座位判断依据的，因此编译器会根据参数的静态类型决定重载那个版本，在很多情况在重载版本是不唯一的，会去选择一个最适合的版本，通过重载方法的优先级进行匹配，

运行时多态（这种多态在程序运行时才能确定下来）：

他是一种动态分派的方式，具体体现就是方法重写，因为重写的方法在通过java反编译后生成的符号引用是相同的，但重写后调用的目标方法并不相同，需要通过invokevirtual指令的多态查找在运行时进行解析，他会把多次相同调用的常量池中类方法的符号引用解析到不同的直接引用上，这个是方法重写的本质，也是多态性的运行时体现。

## ****Java中object常用方法****

1、clone()  
2、equals()  
3、finalize()  
4、getclass()  
5、hashcode()  
6、notify()  
7、notifyAll()  
8、toString()

2.**== 和 equals**

==不可用于比较类型上没有父子关系的两个对象，对于两个对象而言，通过new生成，这两个对象的引用类型通过==比较是不相等的，因为new之后他们指向两个不同的对象，假设这两个对象是String类，“hello”,new String(“hello”)有何区别，对于“hello”这样的直接量，jvm会将他通过常量池来进行管理在编译器就能够确定下来，保证常量池中只有一个，不会产生多个副本，因此他们都将引用常量池中的同一个字符串对象，当使用new String(“hello”),他是在运行期创建的，jvm会先使用常量池来管理“hello”直接量，在调用String类的构造器来创建一个新的String对象，新创建的String对象保存在堆内存中，new String（“hello”），一共生成了两个字符串对象，在一些时候为了能够判断两个引用变量是否相等时，指向判断其值是否相等，这时可以通过equals（）方法。

3**.final**( 类成员变量，实例变量，局部变量，修饰基本类型变量和引用类型变量的区别（只保证这个引用类型变量所引用的地址不会改变，但对象中的成员变量，值完全可以改变），可执行宏替换，修饰final方法（不可被重写，如果不希望子类重写父类的方法可以用，比如Object中的getClass()方法），final类（不可以有子类，子类继承父类会继承它相应的成员变量可以重写方法，有时会导致一些不安全的因素）），而且对于Lambda表达式，匿名内部类和局部内部类在1.8之前不允许访问定义的局部变量，必须通过final修饰，1.8只后可以不进行修饰，但必须按照有final修饰的方法来用，相当于有一个隐式的final

4.**可变类和不可变类，缓存实例的不可变类**（实例中的实例变量是否可以进行修改，通过final设置）

5.内部类（非静态内部类，静态内部类，1.8改进的匿名内部类，1.8的Lambda表达式）

内部类提供了更好的封装，把内部类隐藏在外部类之内，不允许同一个包中的其他类访问该类，内部类成员可以直接访问外部类的私有数据,但外部类不能访问内部类的实现细节，比如内部类的成员变量，必须显示创建对象或类名调用（静态）.可通过外部类.内部类new一个内部类引用。

**对于非静态内部类**（被作为成员内部类定义）不能有静态方法，静态成员变量，静态初始化块，但可以包含普通初始化块，他与外部类的普通初始化块作用相同

**静态内部类（类内部类）**是用static修饰(它的作用是把类成员变成类相关，而不是实例相关，不可修饰外部类，只能用于修饰内部类）的，它属于外部类本身，而不属于外部类的某个对象，但是静态内部类无法访问外部类的实例变量，需要实例或者通过类名，

**匿名内部类**适用于创建那些仅需要一次使用的类，因为创建一个匿名内部类时会立即创建一个该类的实例，设个类会立即消失，它必须继承一个父类或实现一个接口，一般情况下匿名内部类不能显示创建构造器，它只有一个无参的构造器，因此new 接口时括号里面不能有参数，如果继承的是一个父类，可以拥有和父类相似的构造器，具有相同的形参列表例如

某个方法需要完成某一个行为，但这个行为的具体实现无法确定，必须等到执行该方法时才能确定下来。假设：一个方法要要对一个数组进行遍历，但无法确定在遍历数组元素时如何处理这些元素，需要在调用该方法时指定具体的处理行为，此时可以将处理行为当成一个代码块参数进行传递，先有一个实现该方法处理行为的接口和一个抽象方法，然后会有对该处理行为相对应的多个实现类，而将这个接口当做一个参数传入调用类的方法中，然后通过这个接口去调用不同的处理行为。

然而对这个处理行为的接口实现多个实现类可能没有太大意义，因为他们可能只会使用一次，在这种情况下可以使用匿名内部类或者Landma表达式，

而Lambda表达式是匿名内部类的一种简化，它可以不用再去写方法的名字，返回值类型，只要学号是哪个方法括号以及括号内的参数即可，它主要是代替匿名内部类的繁琐语法，而且对于Lambda表达式的目标类型必须是一个函数式接口（值包含一个抽象方法的接口，蛋壳包含多个默认方法，类方法，Runnable,Actionlistener等）。

Lambda表达式与匿名内部类的区别在于，匿名内部类可以实现任意接口的实例，，而且匿名内部类可以为抽象类甚至普通类创建实例，但是Lambda必须是一个函数式接口。

1. String（String.charAt(int index),String.toCharArray(),）,StringBuffer和StringBuilder（+）,String.concat（String str）,

三者都实现了了一个字符串协议接口CharSequence,其中String是一个不可变类，一旦创建一个String对象，包含这个对象的字符序列是不可改变的，要进行改变的时候等同于创建了一个新的String对象，然后用引用地址指向了新的String对象，这样经常改变，每次都会生成生成对象对系统性能造成形象，当内存中无引用对象多了，jvm就会进行垃圾回收，使得运行速度下降，而对于后面两个，每次结果都会对StringBuffer或者StringBuilder本身进行操作，而不是生成新的对象然后改变对象的引用。这两个一个是线程安全的一个是县城不安全的，Stringbuilder是StringBuffer的一个建议替换，用在字符串缓冲区被单个线程使用，因为StringBuffer是一个多线程版本，它在并发使用时效果会优于StringBuilder,但在单线程情况下优先选择Stringbuilder,因为StringBuffer中有一个偏向锁的设置，但也可以在参数中设置禁用偏向锁（-XX:-UseBiasedLocking)或者根据代码逃逸技术，如果判断到一段代码中，堆上的数据不会逃逸出当前线程，那么可以认为这段代码是线程安全的，不必要加锁，例如StringBuffer属于一个局部变量，（线程的安全是以牺牲性能为代价的，所谓线程安全就是多了个加锁，解锁的操作），再者“+’,String.concat（String str）和StringBuilder方法都可以实现字符串连接，但本质+也是通过StringBuilder实现了，因此从代码层面时等效的，但是如果时在一个循环里循环执行”+“操作，通过javap反编译后就可以看出，会生成与循环次数相等的Stringbuilder对象，因此对于字符串连接操作优先选择用StringBuilder来进行操作。

1. ThreadLocalRandom(1.7）Random

ThreadLocalRandom rand = ThreadLocalRandom.current();

Random rand = new Random();然后调用各种nextXxx()

Random专门用于生成一个伪随机数，ThreadLocalRandom是java 1.7新增的一个类，可以在并发访问的环境下使用，来代替Random可以减少减多线程中资源的竞争，使系统具有更好的安全性和较高的性能。

1. BigInteger和BigDecimal

用于处理大整形数据，对于BigDecimal是为解决float,double的精度丢失问题，该类提供了大量的构造器用于创建该对象，宝宝基本数值变量和数字字符串，数字字符数组等，例如new BigDecimal(0.1)来创建一个对象，他的值并不是0.1，只是近似等于0.1，因为0.1无法准确的表示为double，只是表面上看上去是0,1，如果是数字字符串则正好是0.1,因此使用数字字符串会好一些，例如对一个连续整数进行相乘

BigDecimal result = BigDecimal.ONE;

for(int i=2;i<n;i++){

Return result=result.multiply(new BigDecimal(i+””));

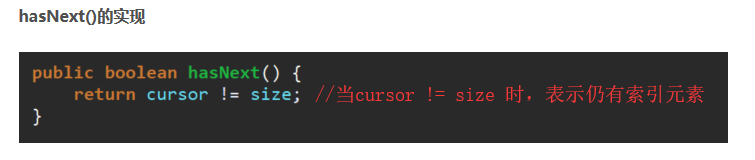
}

1. **JAVA集合(Collection(forEach()), Iterator(forEachmaining()) , foreach, ListIterator ,Spliterator(是一个专门为并行元素设计的课分隔迭代器,Predicate ,Stream ,Set ,List ,Queue ,Map ,Collections工具类，ConcurrentHashMap )**

**Collection**是set，，List,Queue的父接口，其中有add(),isEmpty(),remove（），iterator(),size(),toArray()等多个方法。

**Iterable**接口时Collection的父接口，他在java8新增另一个forEach(Consumer action)遍历集合元素，程序会依次将集合元素传给Consumer的accept(T t)方法，是改接口中唯一的抽象方法，因此可以用Lambda表达式来遍历集合元素。

**Iterator迭代器**，Iterator It = list.iterator(),it.hasNext(),it.next(),it.remove()（从集合中删除上一次next()方法返回的元素或者说是从迭代器指向的collection中移除迭代器返回的最后一个元素）,it.forEachRemaining(Consumer action),是java8新增的默认方法，该方法也可以使用Lambda表达式对集合进行遍历，例如在ArrayList中的Iterator，它有三个成员变量来进行迭代器的遍历进行标识和控制，int cursor；表示下一个元素的索引位置，int lastRet = -1 表示上一个元素的索引位置，int expectedModCount = modCount;预期被修改的次数。



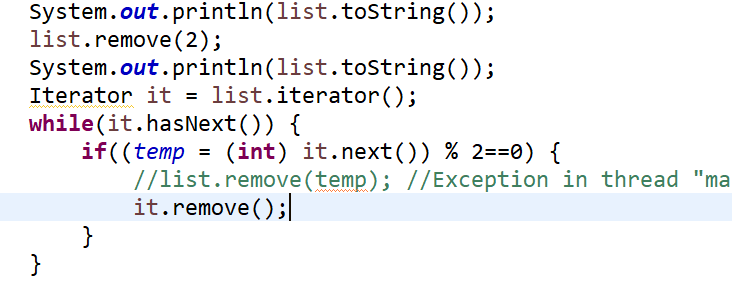




在集合内部维护了一个字段modCount来记录集合被修改的次数，如果几个内部结构发生了改变（add,remove,set操作会对modCount++）

在迭代器内部也维护了一个字段expectedModCount，同样记录当前集合修改的次数，初始化为集合的摸到modCount值，当我们在调用Iterator进行遍历操作时，如果有其他线程修改list会出现modCount！=expectedModCount会抛出并发异常。

就是说在使用迭代器对集合进行删除操作时，可以使用迭代器自身的remove（）进行删除或者不适用迭代在迭代器外部使用集合的remove()方法进行删除，但不能通过该集合的remove（）进行删除会报并发修改异常。



**foreach**的循环迭代集合元素的方法，基于 iteration 和下标遍历的内部实现，该集合不能够被改变，否则将出现并发修改异常。

**ListIterator**接口也用于遍历集合，它提供更强大的遍历措施，可以向前遍历，将集合反向输出。



**Predicate接口**（是一个函数式接口）是java8新增的操作集合，Collection集合新增另一个removeIf(Predicate filter),用于批量删除符合filter条件的所有元素，perdicate有一个test方法，专门用于条件判定。

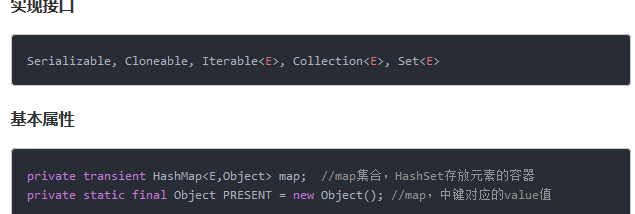
**Stream操作**结合，也是java8新增的操作集合，它是一个支持多个串行和并行聚集操作的元素，XxxStream,每一个流式都有一个builder()来创建对应的流操作，然后可以调用Stream的聚集方法（分为中间的或者末端的）进行聚集操作，且对于聚集操作而言，基本只能执行一次。常用的方法有filter(pridicate p),maptoXXX(),forEach(),count(),max(),min()等，例如对于Collection类型的集合可以直接调用Stream对集合进行过滤判定操作，对于Map集合需要

map.entrySet().stream().filter((ele1)->*checkValue*(ele1)).forEach(ele1->System.***out***.println(ele1.getKey()));

这可用于通过map中的value值来对符合的key值进行输出，也可以保存到另一个集合中。

**Set集合（HashSet,LinkedHashSet,TreeSet,EnumSet）都是线程不安全的**

Set是一种无序，不允许包含可重复的数据容器（如果试图加入两个相同的元素进去，添加操作会失败，add()操作返回false,且新元素不会被加入，并报错），它类似于一个桶，依次将创建的的多个对象丢进去，但它不能够记住元素的添加顺序。

**HashSet** ，**HashSet的实现原理其实是HashMap**，它按照Hash算法（它在于速度比较快，当需要查询某个元素时，hash算法可以直接通过该元素的HashCode值计算出该元素的存储位置，从而快速定位该元素）来存储集合的元素，具有很好的存取和查询功能，它不能保证元素的排列顺序，HashSet不是同步的，因此当多个线程同时秀海HashSet集合时，必须通过代码俩保证同步，且元素的值可以为null,每向HashSet中存入一个元素，HashSet会调用该对象的HashCode（）方法来得到该对象的hashCode值，根据该值去决定该对象在HashSet中的存储位置，因此HashSet集合要判断两个元素是否相等，一时equals()方法比较相等，二是两个对象的HashCode值相等，可以通过重写一个类里面的HashCode和equals来自定义HashSet的存取规则，而他的每一能能存储元素的槽位被称作桶，如果多个元素的HashCode值相等，当equals返回false就需要在一个桶里面放多个元素，HashSet中的元素不建议修改，有可能导致该对象与集合中的其他元素相等，从而导致Hashset无法访问或者无法准确访问。大多数类都是重写了equals()方法的，从而进行的是内容的比较，并且具有自反性，对称性，传递性，一致性，等性质，**当equals()方法被override时，hashCode()也要被override。按照一般hashCode()方法的实现来说，相等的对象，它们的hash code一定相等，**删：更加对象的hashcode找到元素的位置，**如果该hashCode桶产生了冲突，查找的原理和LinkedList一样，时间复杂度为O(n),如果没有产生冲突，时间复杂度为O(1)**。改：查了java的文档，貌似HashSet没有提供直接的修改操作。间接的做法是先删后增。内部实现：

其中PRESENT是一个静态常量，无实际意义。



总结：HashSet底层是通过HashMap实现的，HashMap在1.8之前采用数组+链表的形式，在1.8之后采用数组+链表+红黑树的形式实现，因为Map是无序的，因此HashSet也是无序的，Hahset的方法是借助hashMap的方法来实现的

**LinkedHashSet**他是hashSet的一个子类，因此依然不允许有重复元素，他也是根据HashCode的值来决定元素的存储位置的，但它同时使用链表来维护次序，形成了一种有序的集合，因为需要维护链表的插入顺序，因此没有HashSet的性能高，但在迭代方面有很好的性能。

**TreeSet** 他是SortedSet接口的实现类，Treeset的底层是通过TreeMap实现的，TreeSet是非同步的，线程不安全的它可以确保集合元素处于排序状态，它并不是按照元素的插入顺序进行排序的，而是根据元素的实际值的大小来进行排序的，并且他不是采用hash算法来决定元素的存储位置而是采用红黑树的数据结构来存储集合元素，他在Hashset基础之上还额外有Comparator comparator()(此方法表示如果TreeSet采用定制排序，测返回定制排序所使用的的Comparator，如果采用自然排序，返回null（在默认情况下TreeSet采用自然排序）,还有一些方法返回集合中第一个/最后一个/位于指定元素之前的/之后的/子集合元素等多个方法。

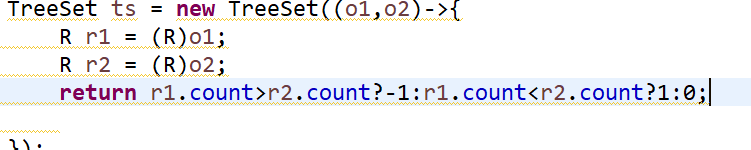
自然排序（升序）：TreeSet会调用集合元素的**compareTo(Object 0)**方法来比较元素之间的大小关系，然后将集合元素按升序排列，（向其中添加元素，若只有一个元素无序实现Comparable接口，若有多个必须实现，否则报异常。且向treeset中添加元素，会将比较的元素类型强制转换成相同的类型，添加对象时，应该添加的是同一个类的对象，也可以实现Comparable接口的compareTo()方法进行自定义，使得Treeset中可以添加多种类型的对象）也就是说如果希望TreeSet正常工作，只能添加同一种类型的对象，比较大小之后，根据红黑树结构找到它的存储位置，不能添加重复元素（也可重写compareTo方法指定他比较的值永远不相等，这样就可以实现相同元素照样能够添加成功，但这样相同元素的引用指向是相同的，当相同元素中一个数值发生改变，其他与其相同的值也会改变，使得数据冗余的出现），并且当treeset中添加一个可变对象后并修改了可变对象的实例变量，这将使得该对象的排序顺序发生改变，但Treeset不会再次调整他的顺序，甚至会使得改变之后的值与其中了另一个对象CompareTo后返回0，并且当试图删除时也会删除失败，因此一般不要修改放在Treeset和Hashset中的集合元素。

定制排序（降序）：可以与comparator对象关联，然后重写compare(T O1,T O2)方法。通过这种方式依然不能向Treeset中添加类型不同的对象。

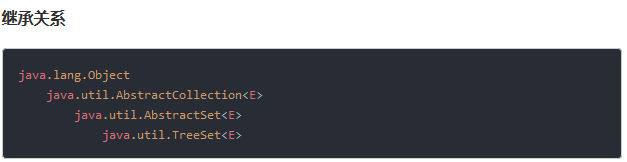
Comparable可以认为是一个**内比较器**，实现了Comparable接口的类有一个特点，就是这些类是可以和自己比较的，至于具体和另一个实现了Comparable接口的类如何比较，则依赖compareTo方法的实现，compareTo方法也被称为**自然比较方法**

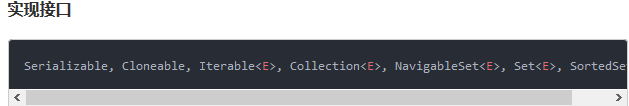
Comparator可以认为是是一个**外比较器，可以实现一些自己想要的比较方式。**

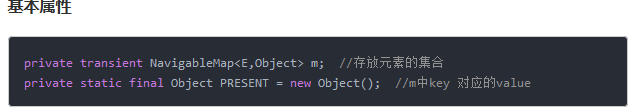
实现Comparable接口的方式比实现Comparator接口的耦合性 要强一些，如果要修改比较算法，要修改Comparable接口的实现类，而实现Comparator的类是在外部进行比较的，不需要对实现类有任何修 改。从这个角度说，其实有些不太好，尤其在我们将实现类的.class文件打成一个.jar文件提供给开发者使用的时候。实际上实现Comparator 接口的方式后面会写到就是一种典型的**策略模式**。



TreeSet的源码分析和红黑树的实现细节









Treeset的底层是通过TreeMap实现的，在构造方法中会创建一个TreeMap实例，用于存放元素，他是有序的，也提供了指定比较器的构造函数，如果没有提供比较器，则会采用key的自然顺序进行比较大小

而且其中的add()和remove（）方法和新增的一些方法都是调用TreeMap的方法进行实现，并且还有相应的克隆方法和针对序列化的写入和读取对象的流方法。

EnumSet：他是一个有序集合，它在内部以为向量的方式进行存储，这样的存储紧凑，高效，不允许加入null,通过allof可以创建一个指定枚举类李所有枚举值的Enumset集合



**List集合（ArrayList和Vector,LinkedList，Arrays）**

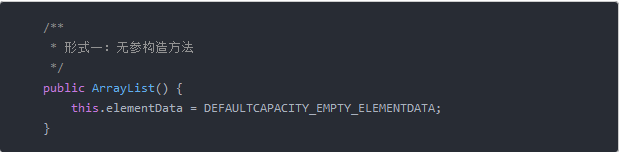
**List**集合代表一个有序，可重复的集合，集合中每个元素都有对应的顺序索引，其中的sort（）排序是根据Compatator外比较器的方式进行元素排序的，通过compare（o1,o2）方法进行，其他方法是在Collection集合的基础上增加了一些根据索引来进行设值和取值的操作。

**ArrayList和Vector(**很多方法都加入了synchronized同步语句，来保证线程安全)

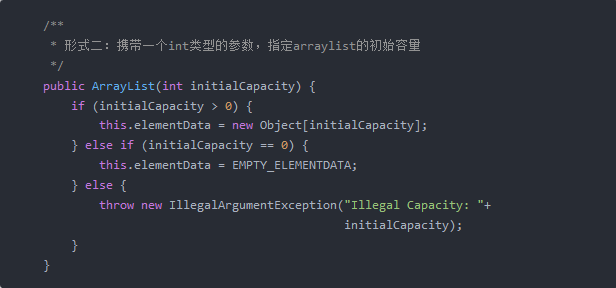
ArrayList是非线程安全的，效率高；Vector是基于线程安全的，效率低 ，他们都是基于数组实现的，因此他们封装了一个动态的，允许再分配的Object()数组，他们都会通过initialCapacity参数来设置该数组的长度，当象他们其中添加元素超出了该数组的长度时，其initialCapacity会自动扩容，对与添加大量元素，可以通过ensureCapacity（int minCapacity)方法一次性增加initialCapacity,这样可以减少重分配的次数，提高性能，如果开始知道他们的大小，可以指定他的初始容量，如果创建一个空的，则数组的默认长度为10。

ArrayList的源码分析：

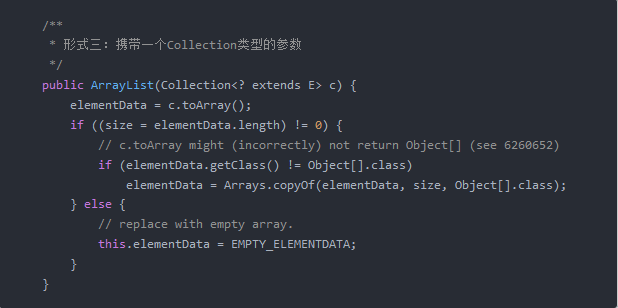




这个时候ArrayList的size为初始值0.



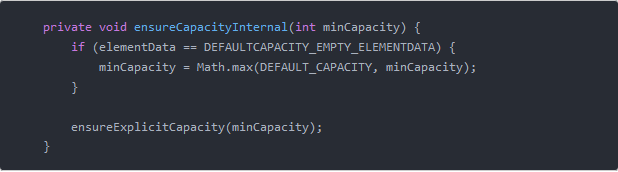
参数initialCapacity为我们所指定的arraylist初始容量，可以看出，方法中对initialCapacity的值进行了一系列判断，当我们所指定的初始容量小于0时无意义，直接抛出非法参数异常。当initialCapacity大于0时，会直接创建一个Object类型的数组，数组的初始大小就是initialCapacity的值。当initialCapacity等于0时，会直接将elementData 指向EMPTY\_ELEMENTDATA空数组。

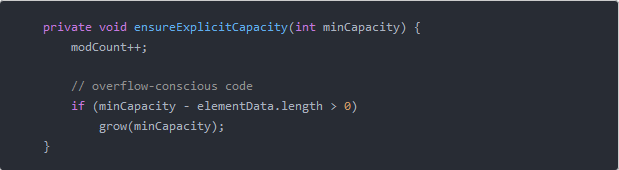


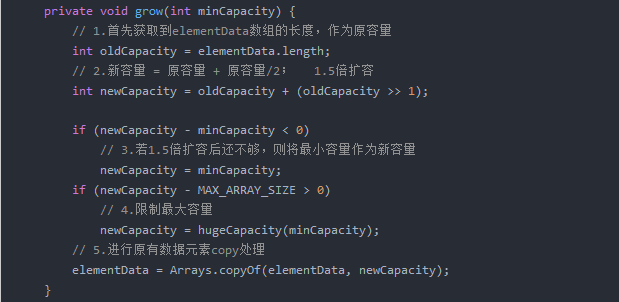
代码中首先将Collection参数通过toArray方法转换成数组，并赋值给elementData，然后对arraylist中的size进行赋值并判断size是否等于0。当size为0时，直接将elementData 指向EMPTY\_ELEMENTDATA空数组。当size不为0时执行copyOf方法。

对于添加一种是直接添加到ArrayList集合的尾部，一种是通过索引来确定下标的位置。



方法中首先调用到ensureCapacityInternal方法，将size+1作为参数传入。我们知道，size用来表示当前arraylist的大小，也就是elementData数组中元素的个数，size+1就是确保数据元素添加成功的最小容量  


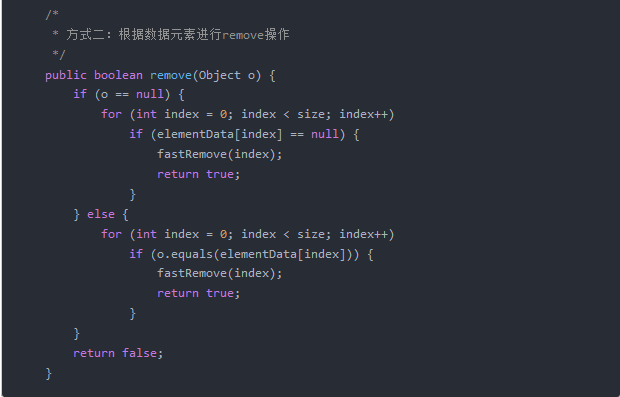
  
方法中首先将变量modCount自增1，modCount是做什么用的呢？其实modCount是用来标记当前arraylist集合操作变化的次数，在fail-fast机制中会有用到这个变量，关于fail-fast机制我们稍后会讲下。接着判断minCapacity - elementData.length 是否大于0，当minCapacity - elementData.length大于0的时候说明当前elementData数组大小不够用，需要扩容，grow方法就是具体的扩容操作

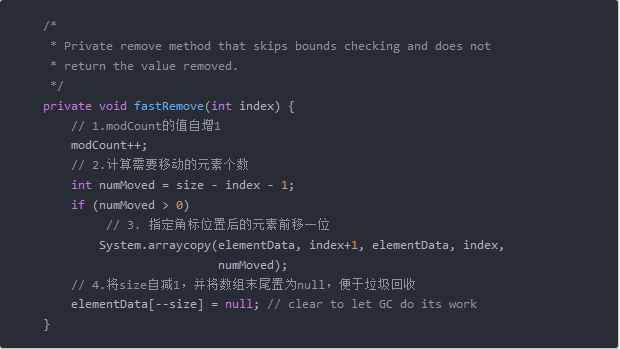


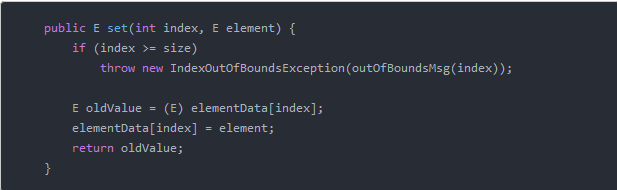
ArrayList是一个1.5倍的扩容，若扩容之后还不够，会将大小设定为这个最小容量。

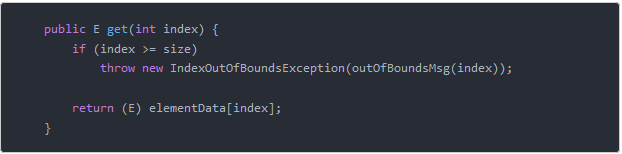
删除操作一种是通过角标index进行删除，一种是直接通过数据元素进行删除。

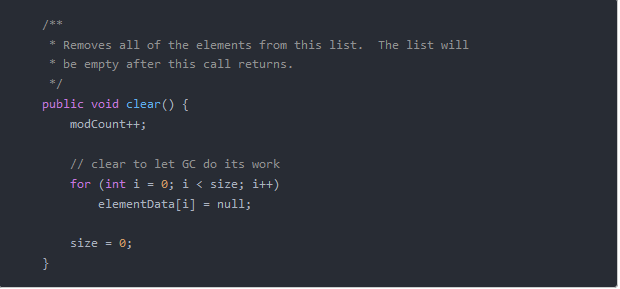


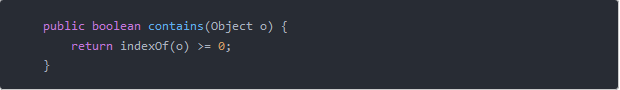


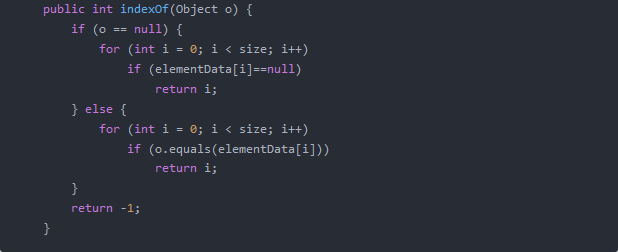












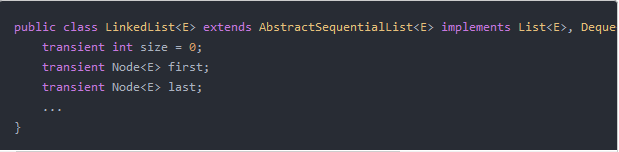
可以看到indexOf方法中对数据元素进行了null判断，两种情况下都进行了遍历操作，只不过两种情况下的if 条件语句不同。当满足条件时，直接将对应的下角标return掉，否则return -1，表示当前arraylist中不存在该元素。

**fail-fast机制**是集合中的一种错误检测机制，我们在操作集合中经常会遇到 java.util.ConcurrentModificationException异常，产生该异常的原因就是fail-fast机制。

这种情况有两种会出现一种情况发生在多线程操作，当a线程正在通过迭代器操作集合mDatas时，同时b线程对mDatas进行添加或者删除元素，会触发fail-fast机制，抛出该异常。另一种情况是在迭代集合mDatas的过程中对mDatas集合进行元素添加或者删除操作时，会触发fail-fast机制，抛出该异常。归根结底就是当我们进行集合数据迭代时，有其他操作修改了当前arraylist的modCount的值，导致modCount != expectedModCount，会抛出该异常。

介绍一种在迭代器中发生该异常的情况，迭代器内部有三个成员变量，分别代表下一个元素的位置，上一个元素的位置，和预期修改的次数，此值为操作集合时的modCount的次数，为注释值，然后再next（），remove（），forEachRemaining(consumer)中都会对预期值与修改值进行判断看是否相等，如果不相等会抛出快速失败相应的异常。

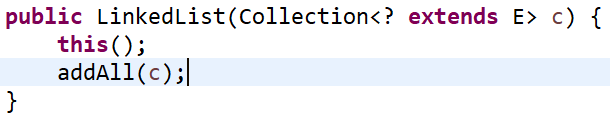
**LinkedList:**是基于双向链表实现的（它既可以当成栈也可以当成一个队列），因此插入和删除操作比ArrayList更高效，因为基于链表实现，因此随机访问的效率要比ArrayList差，增删快，查找慢。是非线程安全的，只是用于单线程环境下，多线程环境下可以采用concurrent并发包下的concurrentHashMap。



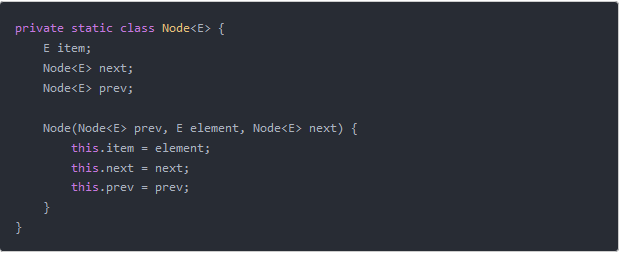
First和last分别代表头结点和尾节点，可以认为是两个额外的指针，分别指着链表的头和尾。

有两个构造函数



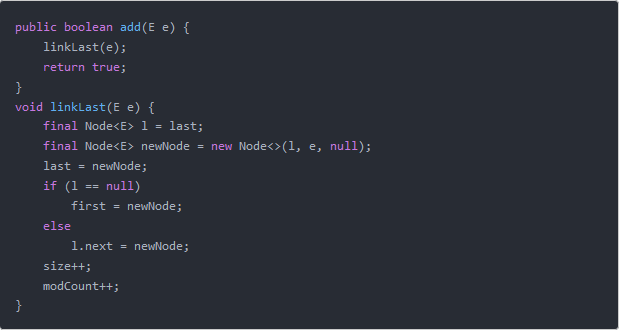


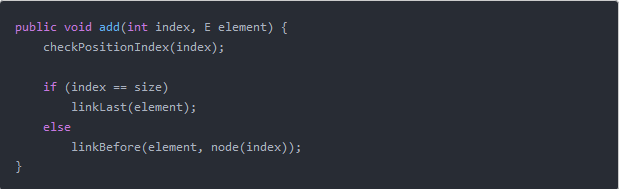
Node是LinkedList类中的一个静态内部类

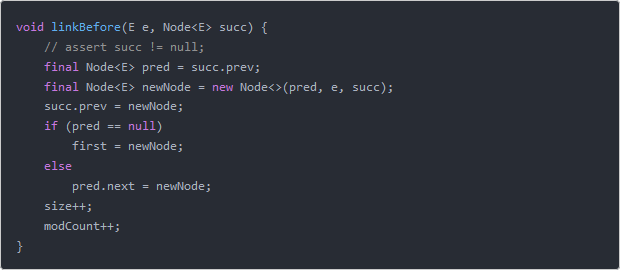


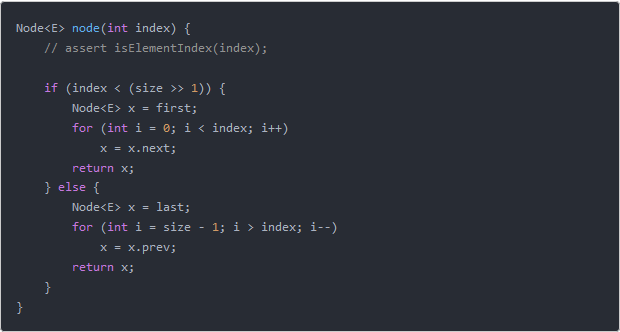
为什么Node这个类时静态的，这跟内存泄露有关，若不适用static修饰，他就是一个普通的内部类，对于普通内部类而言，他在实例化之后，默认是持有外部类的引用的，这就可能造成内存泄漏，使用静态内部类就不会出现。

增加add操作，该方法将添加元素分为2种情况，一种是在集合尾部添加，另一种是在集合中间或者头部添加。



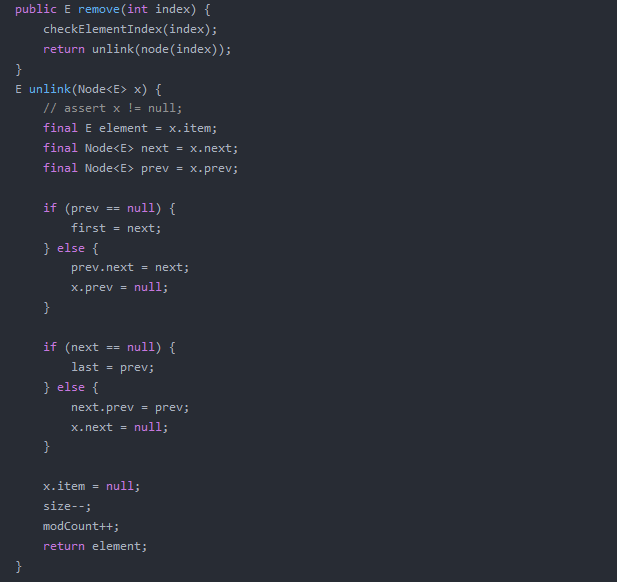


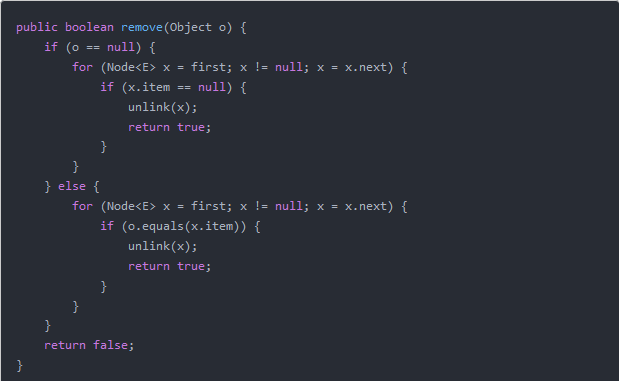




checkPositionIndex用于判断下标是否越界，然后index如果和集合大小相等，则仍然采用尾插法，如果小于则采用linkBefore方法。如果下标在集合大小的前半部分，则从前往后找，或在后半部分，则从后往前找。与ArrayLis相比，因为他是基于数组，因此在查找效率上会比较快，

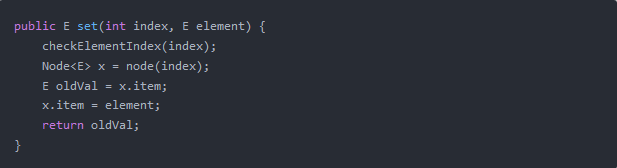
Remove方法，也有两种，一种是通过元素结点，一种是通过形参数据进行。





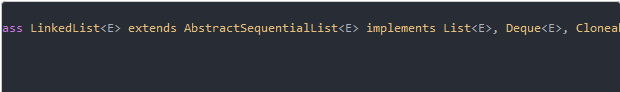
查&改





**Queue集合（ArrayDeque）**

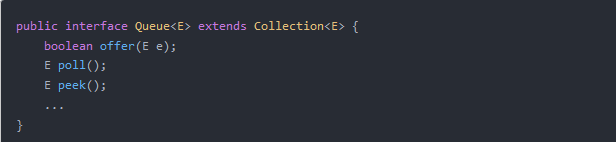
队列的实现就是LinkedList，它实现了队列。



而Deque接口继承于Queue

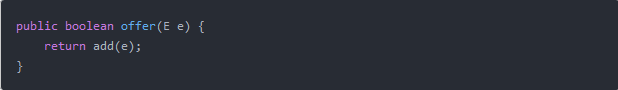


**ArrayDeque**是他的实现类，可以把它实现成栈，也可以把它实现成队列。

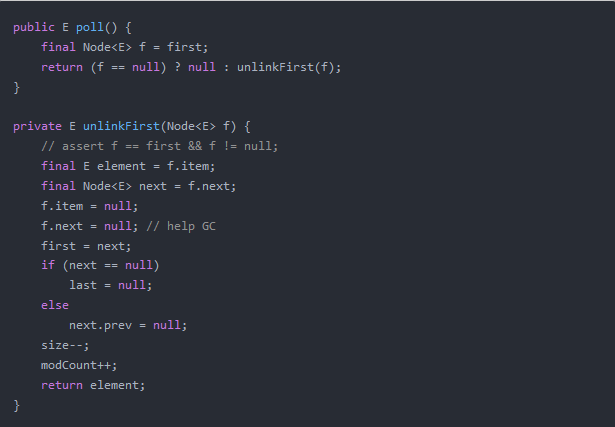


添加对尾元素，删除队头元素，获取队头元素。

**LinkedList对Queue的实现**

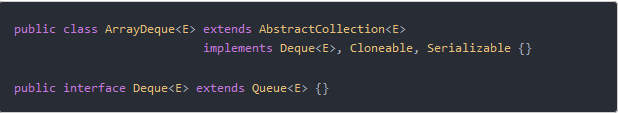


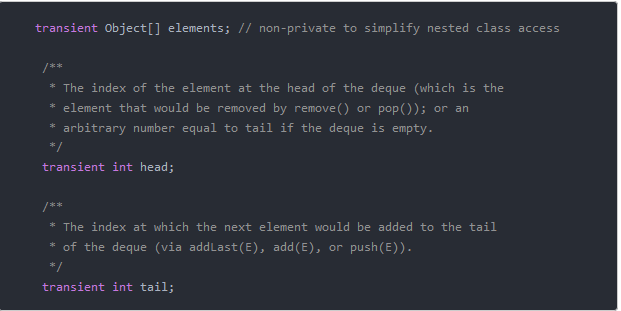
队列的offer(E e)方法实际上是调用了LinkedList的add(E e)，add(E e)已经在最前面分析过了，就是在链表的尾部添加一个元素~



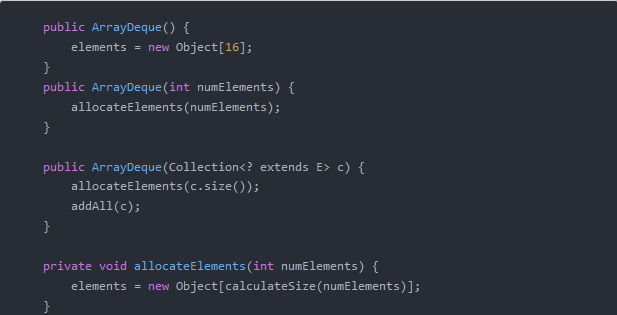
ArrayDeque实现类（栈和队列）

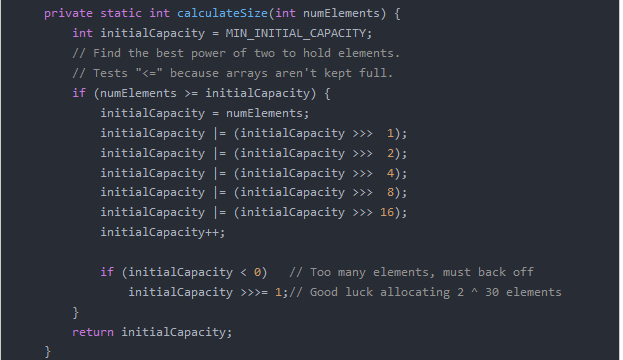
它和ArrayList的实现机制基本相似，他们的底层都采用一个动态的，可重分配的Object[]数组来存储集合元素，当集合元素超出了该数组的容量，系统会在底层重新分配一个Object[]数组来存储集合元素。而且会有两个指针，分别为头指针和尾指针。

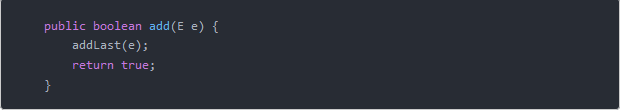


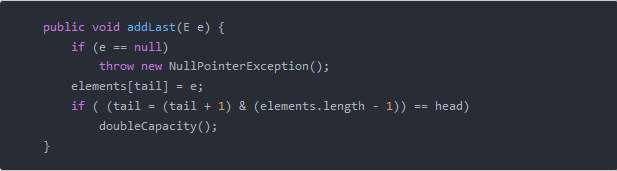


默认开辟16的数组，如果自己指定了大小，那么将调用 allocateElements函数，获取一个 2 的幂次方的大小的容量，其中最小的初始容量是8,若设定的值小于8，则8将作为初始容量，若大于则先将他设为初始容量，然后进行移位和相应的与运算，获取一个 2 的幂次方的大小的容量，或最后他的初始容量小于0,则会将其initialCapacity >>>= 1;以2^30的容量进行设置。

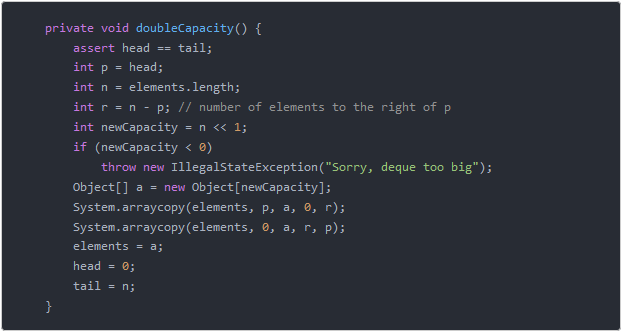




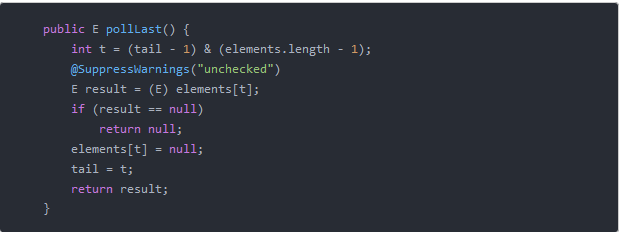




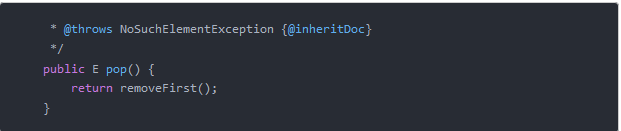
当数组大小刚好为 2 的幂次方时，(tail = (tail + 1) & (elements.length - 1) 为零，也就是说如果head也为0，那么就需要扩容，doubleCapacity 函数，新数组的大小为两倍，使用 System.arraycopy 函数复制，效率极高



pollFirst 获取元素，当然是从 head 位置获取元素，这里需要注意的是，head 如果到了数组末尾，那么又会通过 (h + 1) & (elements.length - 1) 变为 0 了







**Map集合**

**（HashMap,HashTable,CuncurrenthashMap,LinkedHashMap,Properties,SortedMap,TreeMap,WeakHashMap,IdentityHashMap,EnumMap，二叉树查找，平衡二叉树，B树，B+树，红黑树）**

HashMap非线程安全，高效，支持null；

HashTable线程安全，低效，不支持null

SortedMap有一个实现类：TreeMap

Map中常用方法，put(),entrySet(),keySet(),get(key),remove(),values()

,get/set，map.entrySet().Stream().(...)(过滤操作）。

**HashMap与HashTable，**LinkedHashMap，Treemap

**HashMap**是基于哈希表实现的，在java1.8之前采用数组+链表的形式，在1.8之后采用数组+链表+红黑树的形式，其中数组是HashMap的本体，链表时为了解决地址冲突问题，它是一个非线程安全的，适用于单线程，在多线程下使用concurrentHashmap来操作，HashMap因为是基于key的hashCode值来存储value的，所以遍历HashMap不会保证它的顺序和插入时的顺序一致，可以说很大概率这个顺序是不一致的，HashMap允许key为null，但是只允许有一个key为null，他的默认长度是16，每次扩容是2倍的进行，如果扩容后还是不够创建目标长度的数组，将旧数组复制到新数组中，内部通过单链表来实现，根据hashcode计算在数组中的存储位置，判断是否存在相同的元素后，根据情况在相应位置的链表头中插入新元素。

解决哈希冲突的方法主要有开发地址和链地址，HashMap采用了后者，将hashCode相同的记录放在同一个数组位置上，多个hashCode相同的记录被存储在一条链表上，我们知道，链表上的查询复杂的为O(N)，当这个N很大的时候也就成了瓶颈，所以HashMap在链表的长度大于8的时候就会将链表转换为红黑树这种数据结构，红黑树的查询效率高达O(lgN)，也就是说，复杂度降了一个数量级，完全可以适用于实际生产环境

其中会有一个加载因子和相应的阀值，加载因子默认是0.75，阀值用于判断是否需要调整Hashmap容量，等于 容量\*加载因子，如果加载因子越大，对空间的利用更充分，但是查找效率会降低（链表长度会越来越长）；如果加载因子太小，那么表中的数据将过于稀疏（很多空间还没用，就开始扩容了），对空间造成严重浪费。如果我们在构造方法中不指定，则系统默认加载因子为0.75，计算index方法：index = hash & (tab.length – 1)

**空间换时间**：如果希望加快Key查找的时间，还可以进一步降低加载因子，加大初始大小，以降低哈希冲突的概率

Hashtable是在链表的头部添加元素的，而HashMap是尾部添加的，Hashtable是不允许key或者value为null的

java7和java8在实现HashMap上有所区别，当然java8的效率要更好一些，主要是java8的HashMap在java7的基础上增加了红黑树这种数据结构，使得在桶里面查找数据的复杂度从O(n)降到O(logn)，当然还有一些其他的优化，比如resize的优化等

1.7:HashMap的初始桶的数量为16，loadFact为0.75,当桶里面的数据记录超过阈值的时候，HashMap将会进行扩容则操作，每次都会变为原来大小的2倍，直到设定的最大值之后就无法再resize了。Index=hashcode%length进行操作，

**HashTable**：是线程安全的，因为他的大多数方法都是用过同步方法进行操作的，。Size=11,扩容2倍+1，负载因子：0.75,index=index = (hash & 0x7FFFFFFF) % tab.length。他的效率较低，ConcurrentHashmap做了相关优化。

**LinkedHashMap**：LinkedHashMap是HashMap的子类，它将保持记录的插入顺序。

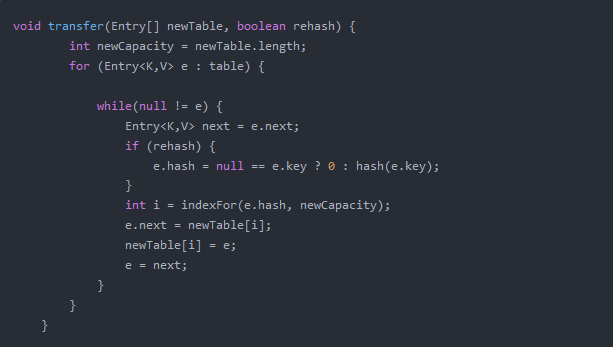
TreeMap:TreeMap实现了SortedMap接口，很明显，他将对插入的记录排序，在遍历TreeMap的时候，得到的是经过排序的记录,然后选定内比较器还是外比较器进行比较接可以了。

**为什么Hashmap是线程不安全的**？

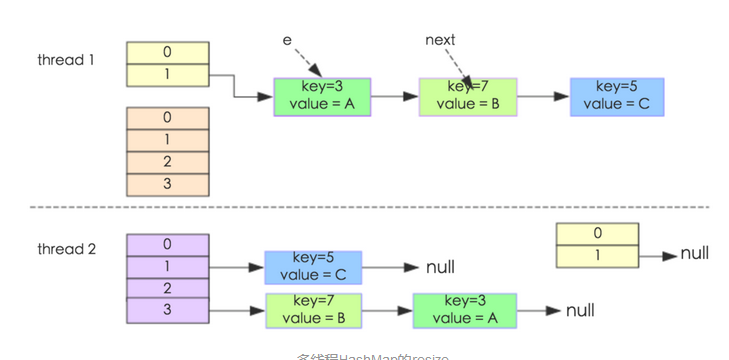
大概的意思就是因为在并发环境下，可能同一时刻有多个线程在操作HashMap，因为HashMap中没有任何措施来保护table，所以在并发环境下多个线程是可以同时操作table的，那么比如在put的时候触发了HashMap扩容，那么在扩容的过程中多个线程的原因可能在某个table的index上会形成一个链表的环，那么此后如果有线程通过get来获取记录的时候，如果刚好这个记录在这个环之后，那么获取记录的线程就会造成死循环

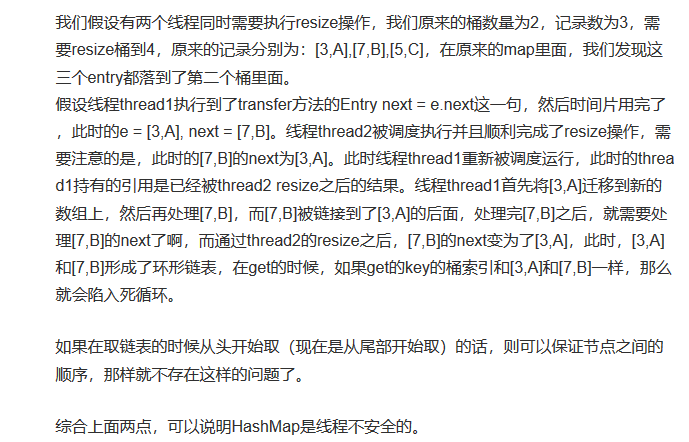
1、put的时候导致的多线程数据不一致。  
这个问题比较好想象，比如有两个线程A和B，首先A希望插入一个key-value对到HashMap中，首先计算记录所要落到的桶的索引坐标，然后获取到该桶里面的链表头结点，此时线程A的时间片用完了，而此时线程B被调度得以执行，和线程A一样执行，只不过线程B成功将记录插到了桶里面，假设线程A插入的记录计算出来的桶索引和线程B要插入的记录计算出来的桶索引是一样的，那么当线程B成功插入之后，线程A再次被调度运行时，它依然持有过期的链表头但是它对此一无所知，以至于它认为它应该这样做，如此一来就覆盖了线程B插入的记录，这样线程B插入的记录就凭空消失了，造成了数据不一致的行为。

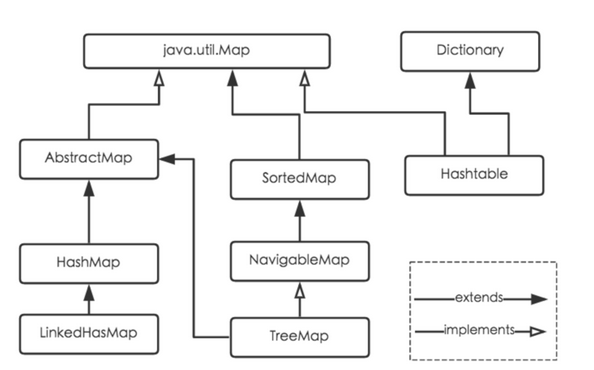
2、另外一个比较明显的线程不安全的问题是HashMap的get操作可能因为resize而引起死循环（cpu100%）



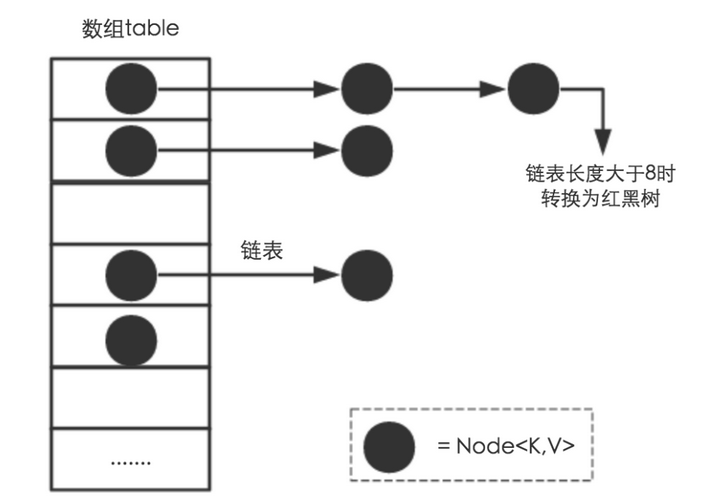
这个方法的功能是将原来的记录重新计算在新桶的位置，然后迁移过去，







源码分析：

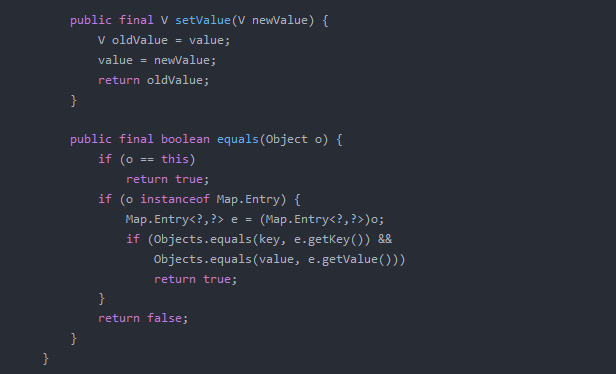


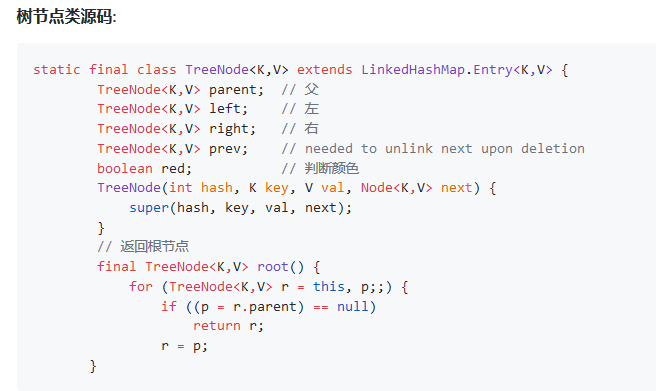


HashMap的哈希桶table的大小必须为2的n次方

链表结点数据结构：

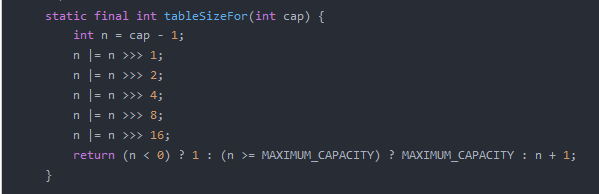






HashMap的构造函数

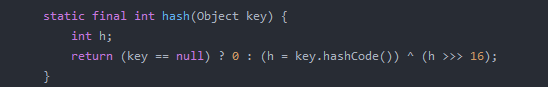




该方法返回与给定数值最接近的2的N次方的值。

**Hashmap如何确定记录的table位置？**

想要找到记录，首先要确定记录在table中的index，然后才能去table的index上的链表或者红黑树里面去寻找记录，下面的方法hash展示了HashMap是如何计算记录的hashCode值的方法：



hash算法只是计算出了HashCode值，然后通过hashCode定位index，也就是对hashCode取模（hashcode%length）,length是table的长度，但是取模是一个复杂的计算过程，因为设定table的长度的必然是2的n次方，这是关键，hashmap通过设定table的长度为2^n，在取模的时候就可以通过hashCode & (length -1)来得到相应的index值，在length总是2的n次方的前提下，上面的算法等效于取模，但是效率会有所提升。

做完了前期的准备工作，现在是hashMap的方法实现部分。

Put方法

· 首先判断table是否为null或者长度为0，如果是，那么调用方法resize来初始化table，resize的细节将在下文中进行分析，这个方法用来对HashMap的table数组扩容，它将发生在初始化table以及table中的记录数量达到阈值之后。

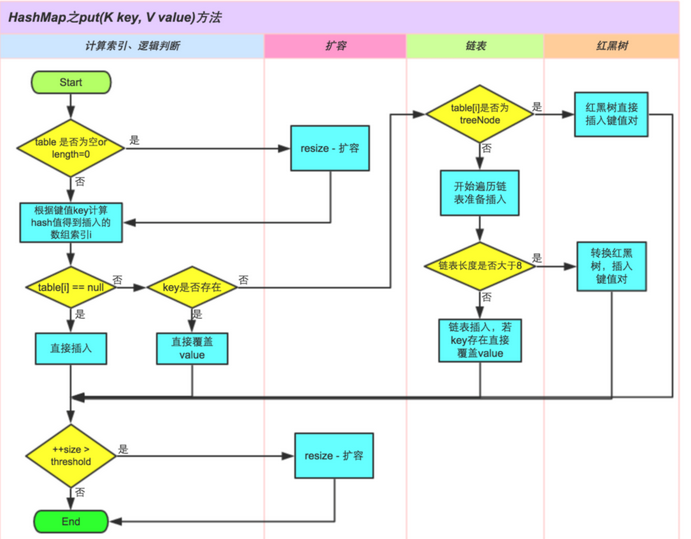
· 然后计算记录的hashCode，以及根据上文中提到的方法来计算记录在table中的index，如果发现index未知上为null，则调用newNode来创建一个新的链表节点，然后放在table的index位置上，此时表面没有哈希冲突。

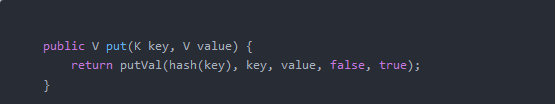
· 如果table的index位置不为空，那么说明造成了哈希冲突，这时候如果记录和index位置上的记录相等，则直接覆盖，否则继续判断

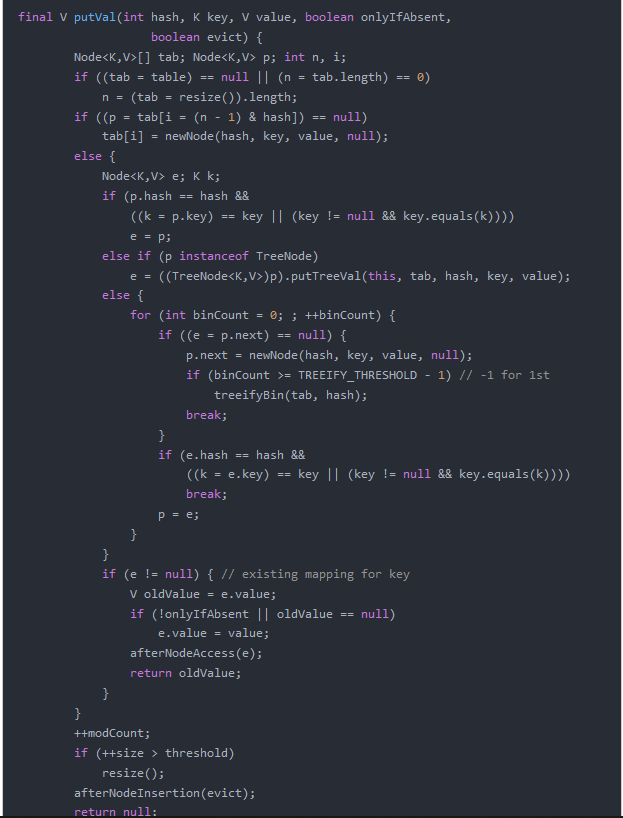
· 如果index位置上的节点TreeNode，如果是，那么说明此时的index位置上是一颗红黑树，需要调用putTreeVal方法来将这新的记录插入到红黑树中去。否则走下面的逻辑。

· 如果index位置上的节点类型不是TreeNode，那么说明此位置上的哈希冲突还没有达到阈值，还是一个链表结构，那么就根据插入链表插入新节点的算法来找到合适的位置插入，这里面需要注意的是，新插入的记录会覆盖老的记录，如果这个新的记录是首次插入，那么就会插入到该index位置上链表的最尾部，这里面还需要一次判断，如果插入了新的节点之后达到了阈值，那么就需要调用方法treeifyBin来讲链表转化为红黑树。

· 在插入完成之后，哈希桶中记录的数量是否达到了哈希桶设置的阈值，如果达到了，那么就需要调用方法resize来扩容。







扩容，就是新申请一个较大容量的数组table，然后将原来的table中的内容都重新计算哈希落到新的数组table中来，然后将老的table释放掉。这里面有两个关键点，**一个是新哈希数组的申请以及老哈希数组的释放**，另外一个是**重新计算记录的哈希值以将其插入到新的table中去**。首先第一个问题是，扩容会扩大到多少，通过观察上面的代码可以确定，每次扩容都会扩大table的容量为原来的**两倍**，当然有一个最大值，如果HashMap的容量已经达到最大值了，那么就不会再进行扩容操作了。第二个问题是HashMap是如何在扩容之后将记录从老的table迁移到新的table中来的。上文中已经提到，**table的长度确保是2的n次方，那么有意思的是，每次扩容容量变为原来的两倍，那么一个记录在新table中的位置要么就和原来一样，要么就需要迁移到(oldCap + index)的位置上**

假设原来的table大小为4，那么扩容之后会变为8，那么对于一个元素A来说，如果他的hashCode值为3，那么他在原来的table

上的位置为(3 & 3) = 3,那么新位置呢？(3 & 7) = 3,这种情况下元素A的index和原来的index是一致的不用变。再来看一个

元素B，他的hashCode值为47，那么在原来table中的位置为(47 & 3) = 3，在新table中的位置为(47 & 7) = 7，也就

是（3 + 4），正好偏移了oldCap个单位。

**那么如何快速确定一个记录迁移的位置呢？**因为我们的计算方法为:(hashCode & (length - 1))，而扩容将导致(length - 1)会新增一个1，也就是说，hashCode将会多一位来做判断，如果这个需要新判断的位置上为0，那么index不变，否则变为需要迁移到(oldIndex + oldCap)这个位置上去

3 ： 1147： 101111

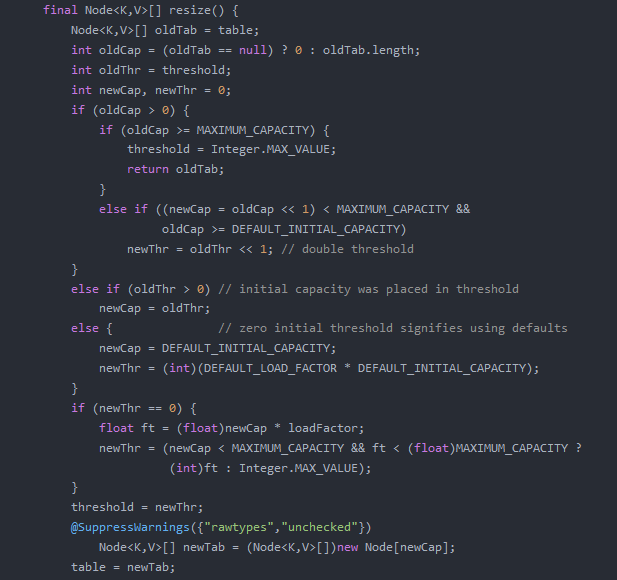
在table的length为4的前提下：

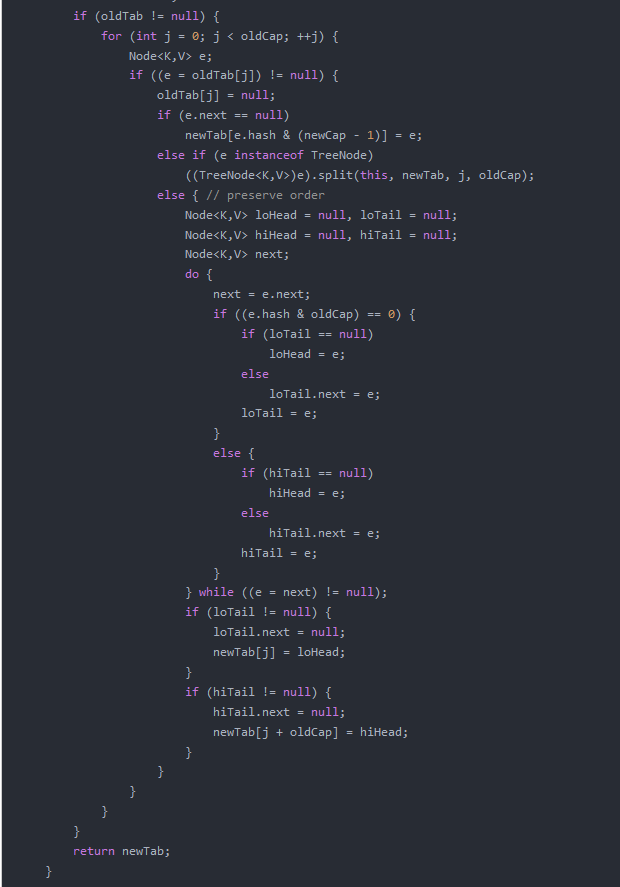
3-> 11 & 11 = 347-> 000011 & 101111 = 3

在扩容后，length变为8：3-> 011 & 111 = 347-> 10111 & 00111 = 7

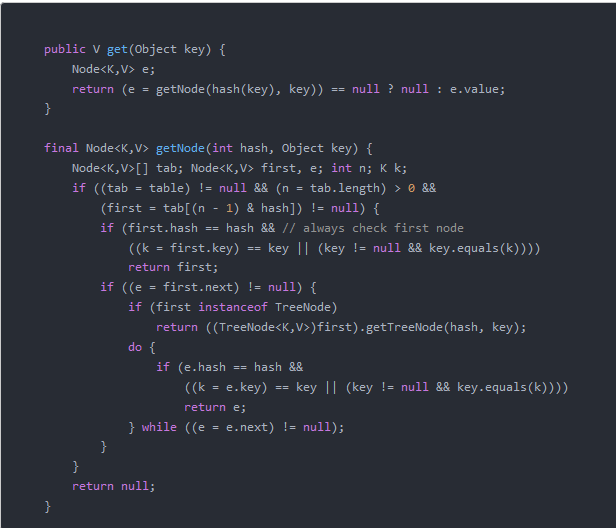
对于3来说，新增的参与运算的位为0，所以index不变，而对于47来说，新增的参与运算的位为1，所以

index需要变为(index + oldCap)



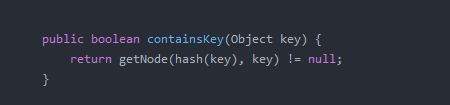


获取get()方法

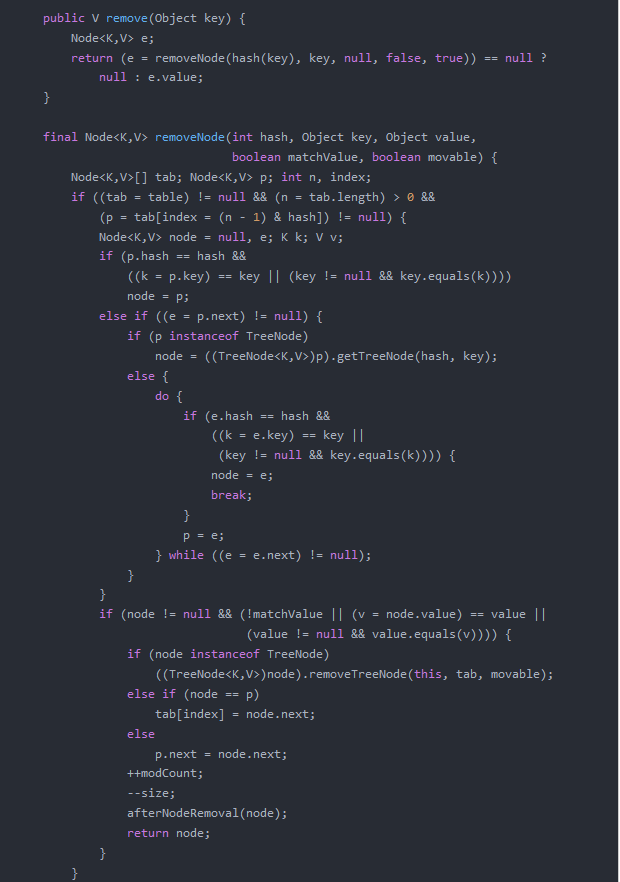


首先会获得当前table的一个快照，然后根据需要查找的记录的key的hashCode来定位到table中的index，如果该位置为null，说明没有没有记录落到该位置上，也就不存在我们查找的记录，直接返回null。如果该位置不为null，说明至少有一个记录落到该位置上来，那么就判断该位置的第一个记录是否使我们查找的记录，如果是则直接返回，否则，根据该index上是一条链表还是一棵红黑树来分别查找我们需要的记录，找到则返回记录，否则返回null。

**如何判断HashMap中是否有一个记录的方法？**



删除remove()

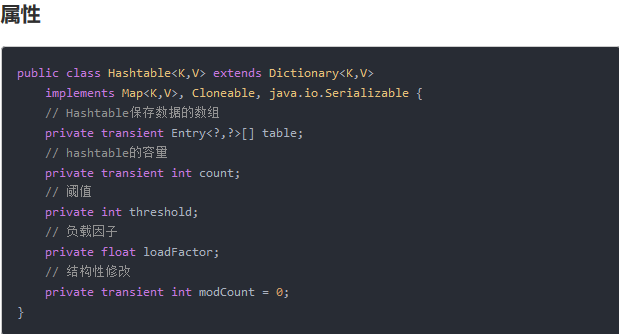


首先，通过记录的hashCode来找到他在table中的index，因为最后需要返回被删除节点的值，所以需要记录被删除的节点。当然记录被删除的节点也是有意义的，比如对于table中的index位置上为一条链表的情况来说，我们只需要记住需要删除的Node，然后真正删除的时候就可以只需要操作该node就可以了

**HashTable**源码分析

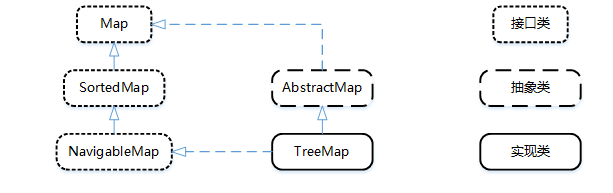
HashTable底层与java1,8之前一样，都是采用数组+链表的形式存在的，他的初始size为11，扩容机制是2倍+1的扩容，负载因子为0.75，计算index的方法index = (hash & 0x7FFFFFFF) % tab.length，它对一些常规的方法都采用重量级锁定进行同步处理，虽然这样保证了线程安全，但是由于加锁和释放锁锁带来的性能消耗，使得其整体的性能不好。Hashtable是不允许key或者value为null的，Hashtable是在链表的头部添加元素的，而HashMap是尾部添加的

在性能上HashTable容器在竞争激烈的并发环境下表现出效率低下的原因是所访问的Hashtable的线程都必须竞争同一把锁，而ConcurrentHashMap是对他的一种优化，采用锁分段技术，指的一个容器里有多把锁，每一把锁用于锁容器其中一部分数据，这样当多线程访问容器里不同数据段的数据时，线程就不会出现竞争资源的情况，这样可以有效提升高并发访问效率。

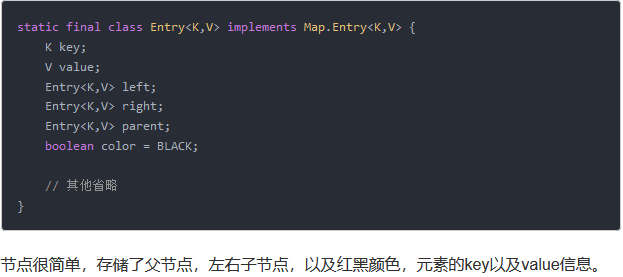


**Linkedhashmap**:是使用双向链表来维护key-value对的次序的，因为他需要维护元素的插入顺序，因此性能低于HashMap的性能，但因为它是以链表来维护维护内部顺序的，所以在迭代访问Map里的全部元素时将会有较好的性能。（可通过他实现一个LRU），其底层是散列表和双向链表，允许为null,不同步，装载因子和初始容量对其影响较大，结构性的修改会影响遍历的顺序，

**TreeMap**:

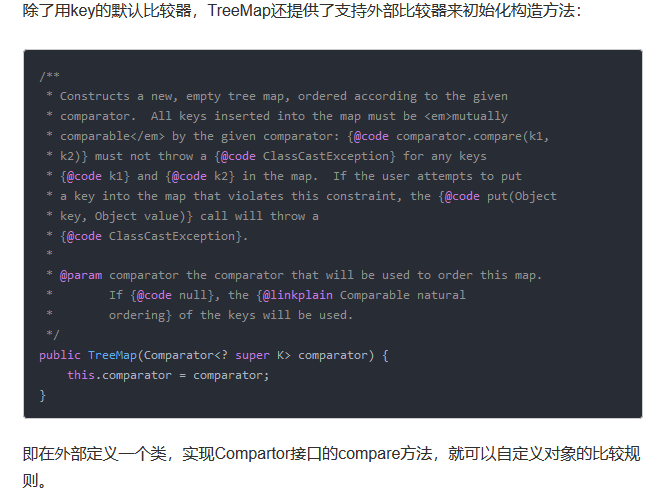


TreeMap采用红黑树的数据结构来实现，树节点Entry实现了Map.Entry，采用内部类的方式实现，并且set和map的关系十分密切，java源码就是先实现了Hashmap，TreeMap等集合，然后通过包装一个所有的value都为null的map集合实现了set集合类。他的方法增加了访问前一个，后一个，第一个，最后一个key-value对的方法，并有集合截取子Treemap的方法，和Treeset类似。









红黑树的删除分两步：  
（1）以二叉查找树的方式删除节点。  
（2）恢复红黑树的性质。

删除分为四种情况：

1. 树只有根节点：直接删除即可。  
   2、待删除节点无孩子：直接删除即可。  
   3、待删除节点只有一个孩子节点：删除后，用孩子节点替换自己即可。  
   4、待删除节点有两个孩子：删除会复杂点

ConcurrentHashMap:

采用锁分段技术，指的一个容器里有多把锁，每一把锁用于锁容器其中一部分数据，这样当多线程访问容器里不同数据段的数据时，线程就不会出现竞争资源的情况，这样可以有效提升高并发访问效率，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一段数据的时候，其他段的数据也能被其他线程访问。

在1.8之前，他是通过Segment数组结构（数组+链表）+HashEntry数组结构组成，segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据，一个ConcurrentHashMap里面包含一个segment数组，一个segment守护一个HashEntry数组，当对HashEntry数组的数据进行修改时，必须首先获得与它对应的segment锁。并且它的segmnet的数组长度ssize是通过ConcurrentLevel（默认值是16）计算得出的，为了实现按位与的散列算法来定位segment数组的索引，必须保证segment数组的长度是2^n，必须计算出一个大于等于ConcurrentLevel的最小的2^n来作为segment数组的长度，并且segment的最大值为2^16次方，初始的偏移量ssize=1,因为concurrentLevel的初始值是16，因此他需要向左按位移动4次，为4，段偏移用于定位参与散列运算的位数，等于32-sshift，等于28，用32是因为ConcurrentHashMap里的hash()方法输出的最大数是32位的。

定位segment：他首先会通过hash算法对元素的hashcode进行一次再散列，目的是减少冲突，然后通过**（hash>>>segmentShift）$segmentMark** 得到定位结果，段偏移的默认值时28，段掩码为15；他有3中操作，get操作，put操作，size操作，

定位Segment使用的是元素的hashcode通过再散列后的值，目的是避免两次散列后的值一样，虽然元素在segment里散列开了，但是却没有在HashEntry里散列开。**（hash&(tab.length-1）**;

**Get操作**：他会先经过一次再散列，然后使用这个散列值通过散列运算定位到Segment，再通过散列算法定位到元素，get操作的整个过程不需要加锁，除非读的值为空才进行加锁重读，它不加锁是因为get方法里将要使用的共享变量都定义成volatile类型，定义成volatile变量，能够在线程之间保持可见性，能够被多线程同时读，并且保证不会读到过期的值，但是只能在单线程进行写操作，之所以不会读到过期的值，是因为根据java内存模型的happen before原则，对valatile字段的写入操作先于读操作，不管怎样，最终都能拿到最新的值。

**Put操作**：需要对共享变量进行写入操作，所以为了线程安全，在操作共享变量时必须加锁，首先要先定位到segment，然后再segment中进行插入操作，插入操作分两步完成，第一判断是否需要对segment里的HashEntry数组进行扩容，第二步定位添加元素的位置，然后将其放在HashEntry数组中。

1. 是否需要扩容

在插入元素前会先判断Segment里的HashEntry数组是否超过容量，如果超过阀值，在进行扩容，相比HashMap的扩容是在插入元素后进行扩容，这种情况下，如果扩容之后再也没有元素插入，就相当于进行了一次无效的扩容

1. 如何扩容

他的扩容思想基本HashMap的一样，都是进行2倍的扩容，但他不会对整个容器进行扩容，只是会给相应的segment进行扩容。

Size()操作：会先尝试两次去将各个segment中的count进行相加，如果在此过程中人一个操作的modCount改变了，那么就会和预期修改次数进行比较，看是否发生变化若没有发生变化，则所得到的的Size()是正确的，却发生了变化，则会将这三个操作统统锁住，然后再进行操作，因为这样的造作方式效率态低。

在1.8之后舍弃了Segment，并且大量采用synchronized，以及CAS无锁操作以保证ConcurrentHashMap操作的线程安全性，并且它的底层数据结构改变为：数组+链表+红黑树的数据形式。

WeakHashMap:与hashmap用法相似，HashMap的key保留了对实际对象的强引用，这就意味着只要该Hashmap不销毁，该Hashmap的所有key所引用的key对象就不会累垃圾回收，而WeakHashMap的Key只保留了实际对象的若引用，就是说WeakHashMap对象的key所引用的对象没有被其他强引用变量所引用，则这些key所应用的对象可能被垃圾回收，他也能删除这些key所对应的key-value对。

IdentityHashMap：

EnumMap:

Collections工具类:其中的SynchronizedXxx（）方法，可以将集合包装成线程同步的方法。

1.Arrays.sort()  
该算法是一个经过调优的快速排序，此算法在很多数据集上提供N\*log(N)的性能，这导致其他快速排序会降低二次型性能。

2、Collections.sort()  
该算法是一个经过修改的合并排序算法（其中，如果低子列表中的最高元素效益高子列表中的最低元素，则忽略合并）。此算法可提供保证的N\*log(N)的性能，此实现将指定列表转储到一个数组中，然后再对数组进行排序，在重置数组中相应位置处每个元素的列表上进行迭代。这避免了由于试图原地对链接列表进行排序而产生的n2log(n)性能。

1. 泛型原理，使用场景，优缺点

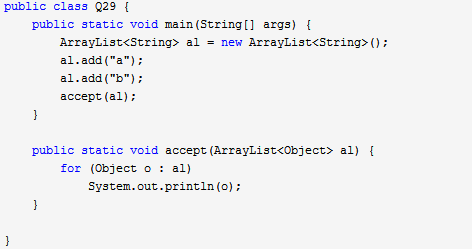
原理：泛型的实现是靠类型擦除技术，类型擦除是在编译期完成的，在编译期，编译器会将泛型的类型参数都擦除成它的限定类型，如果没有则擦除为object类型之后在获取的时候再强制类型转换为对应的类型。

使用场景：参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口和泛型方法。

优点：

1. 类型安全
2. 消除强制类型转换
3. 潜在的性能收益

缺点：在性能上不如数组快。



上面的代码编译就不会通过，云因在于类型擦除，java的泛型机制是在编译级别实现的，编译器生成的字节码在运行期间并不包含泛型的类型信息，在编译之后，泛型将会被擦除成他的限定类型，如果没有则擦除为object类型之后在获取的时候再强制类型转换为对应的类型，他们的类型信息对JVM是不可见的，因此一般发现类型不一致，就会报错。与数组比较：如果两个泛型类型有相同的参数，他们的继承关系是依据其类型，如ArrayList<String>和Collection<string>，但对于数据，他们在运行期间知道每个元素的类型且强制他们的元素为该类型。

1. IO,NIO.1,NIO.2,AIO

在阻塞模式下，若从网络流中读取不到指定大小的数据量，阻塞IO就在那里阻塞着。比如，已知后面会有10个字节的数据发过来，但是我现在只收到8个字节，那么当前线程就在那傻傻地等到下一个字节的到来，对，就在那等着，啥事也不做，直到把这10个字节读取完，这才将阻塞放开通行。

在非阻塞模式下，若从网络流中读取不到指定大小的数据量，非阻塞IO就立即通行。比如，已知后面会有10个字节的数据发过来，但是我现在只收到8个字节，那么当前线程就读取这8个字节的数据，读完后就立即返回，等另外两个字节再来的时候再去读取。

从上面可以看出，阻塞IO在性能方面是很低下的，如果要使用阻塞IO完成一个Web服务器的话，那么对于每一个请求都必须启用一个线程进行处理。而使用非阻塞IO的话，一到两个线程基本上就够了，因为线程不会产生阻塞，好比一下接收A请求的数据，另一下接收B请求的数据，等等，就是不停地东奔西跑，直接到把数据接收完了。

虽然说，非阻塞IO比阻塞IO有更高的性能，但是对于开发来的，难度就成数倍递增了。由于是有多少数据就读取多少数据，这样在读取完整之前需要将已经读取到的数据保存起来，而且需要与其他地方来的数据隔离开来不能混在一起，否则就不知道这数据是谁的了

IO:安类型主要分为字符流和字节流，

按功能分为处理流和节点流，对于处理流而言，不能指定目的地和源的流，需要刘外一个流来创建你这个流，他就好比水管，而流就是水管中的水，节点流能指定目的地和源的流，因此他能够直接对文件进行操作。

流式常用的有：

**InputStrem/outputStream**：

FileInputStream，FileOutputStream（节点流），缓冲流可通过这个获得的结点流，然后指定一个缓冲数组，来对缓冲流进行模拟。

DataInputStream,DataOutputStream（处理流）,

ObjectInputStream,ObjectOutputStream(处理流）.

StringBufferInputStream（节点流）字符串缓冲流

Reader/Writer:

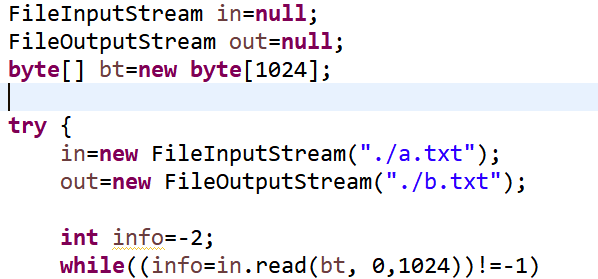
FileReader,FileWriter(结点流）,

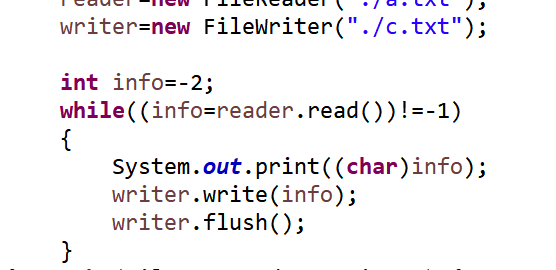
BufferedReader,BufferedWriter（处理流）,

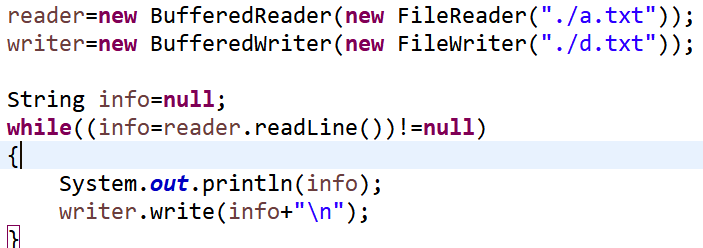
StringReader,StringWriter（节点流）,

InputStreamReader,OutputStreamWriter（字节字符转换流）先将字节流转换成字符流然后再通过字符流进行使用。

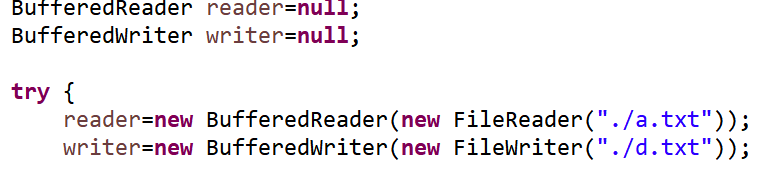
非流式：file,RandomAccessFile.











输入流输出流都是阻塞式的输入，输出，传统的输入流，输出流都是通过字节的移动来处理的，即使不直接处理字节流，底层也是通过字节处理的，因此面向流的输入输出系统一次只能处理一个字节，他们的效率不高；例如BufferedReader读取输入流中的数据时，如果没有读到有效数据，程序将在此处阻塞该线程的执行。

****Java序列化与反序列化是什么？为什么需要序列化与反序列化？如何实现Java序列化与反序列化****

**序列化机制使得对象可以脱离程序的运行而独立存在，而对象序列化的目标是将对象保存到磁盘中，或允许再网络中直接传输，它将java对象转换成与平台无关的二进制流存储在磁盘进行持久化或者在网络上传输后通过反序列化得到原来的java对象。一种是通过实现Serializable（不需要做什么），一种是实现Externalizable(他要实现writeExternal（out）和readExternal（in）两个方法，并提供具体的实现，**

NIO.1（1.4）

NIO采用内存映射文件的方式来处理输入/输出，新Io将文件或文件的一段区域映射到内存中，这样就可以像访问内存一样来访问文件。

Channel（通道），Buffer(缓冲）是新IO的两个核心对象。

**Channel**是对传统的输入/输出系统的模拟，在新IO系统中所有的数据都需要通过通道传输，他与阻塞IO的区别是他提供了一个map()方法，通过该方法可以直接将一块数据映射到内存中，因此也可以说NIO是面向块的处理，他可以直接将指定文件的部分或全部文件直接y映射到Buffer，但是不能直接访问Channel中的数据，Channel只能与Buffer进行交互，就是说向从Channel中取出数据，必须先用Buffer从channel中取出数据，然后让程序从Buffer中取出数据，写入也是一样。

Java为Channel接口提供了用于支持线程通信的管道，用于支持的Tcp和UDP通信的通道，和处理文件的FileChannel。

它的创建不是用构造器，是用传统的节点流和非流式RandomAccessFile（）的getChannel()方法来返回对应的Channel，不同的结点流返回过的的Channel不一样。

它常用的方法有map(),read(),write()方法，map()方法用于将Channel对应的部分或全部数据映射成ByteBuffer，他的方法签名：MappedByteBuffer map(FileChannel.MapMode,long position,long size),第一个用于规定读写类型，后面是需要映射到Buffer的部分，read(),write()都是一些重载形式，用于在Buffer中读取和载入数据,

**Buffer**可以被理解成一个容器，他的本质是一个数组，发送到Channel中的所有对象都必须首先放到Buffer中，而从Channel中读取的数据也必须先放到Buffer中。他是一个抽象类，子类有XxxBuffer，常用的是ByteBuffer，CharBuffer,而且对于ByteBuffer而言，它还有一个子类MappedByteBuffer，他表示将磁盘文件的部分或全部内容映射到内存中后得到结果，该对象有Channel的map()对象返回。

**它有三个概念：容量（capacity)，界限（limit)和位置（position)**

**容量：他是缓冲区Buffer的最大容量，可以通过allocate（int capacity)创建大小**

**界限：规定位于limit后的数据不可读，不可写。**

**位置：与IO中的记录指针类似，当使用Buffer从Channel中读取数据时，position恰好等于已经读到了多少数据**。

当放入数据之前，他的position=0,limit=capacity，当放入数据之后每放入一个数据，position会相应的向前移动，当装入结束后，调用Buffer的filp()方法，该方法将limit设置为position所在的位置，并将position设置为0，为输出数据做准备，当输出结束后，Buffer会调用clear()方法，它不会清空Buffer中数据，它仅仅是将position设置为0，将limit设置为capacity，为再次向Buffer中装入数据做准备。

对于Buffer而言他的主要作用就是装入数据和输出数据，他有两个方法**filp(),clear()**，filp()为从Buffer中取出数据做好准备，而clear()为再次向Buffer（）中装入数据做好准备，并且除了一些基本的定位和判定操作外，还提供了put(),get()方法，用于向Buffer中放入数据和从buffer中取出数据，这样Buffer既支持对单个数据的访问也支持对批量数据的访问，在这两种情况下访问Buffer中的数据，为为相对和绝对：

相对：从Buffer的当前position处开始读取或写入数据，然后将位置position的值按处理元素的个数增加。

绝对：按照索引进行读取和写入，不会影响position。

**CharSet**：用于将Unicode字符串映射成字节序列以及逆映射操作，它是将明文的字符序列编码（Encode）成二进制字节序列和将二进制字节序列解码(decode)成明文的字符序列的过程,用来处理字节序列和字符序列之间的转换关系，通过forName（“GBK”）指定他的字符集，然后通过newEncode(),newDecode（）来创建相应的字符编码器CharSetEnCoder和字符解码器CharSetdecoder，然后调用他的个子的encode（），decode()方法实现字节与字符的相互转换。

**FileLock文件锁**：如果多个运行的程序需要并发修改同一个文件，程序之间需要某种机制来进行通信，使得文件锁有效地阻止多个进程并发修改同一个文件，他通过显式的使用lock()和trylock()(定时锁替代内部锁机制）方法可以获得文件锁对象FileLock,从而锁定对象，通过release()方法释放文件锁，当Lock()锁定文件的时候，若无法锁定，程序将一直阻塞(他是阻塞式的），tryLock（long position,long size,boolean shared）他是尝试锁定文件，如果获得文件锁，该方法返回文件锁，否则返回null,该方法是非阻塞式的加锁。当shared为true时，说明他是一个共享锁，它将允许多个进程来读取文件，但阻止其他进程获得该文件的排他锁，当shared为false时，表明该锁是排他锁，他将锁住对该文件的读写，可以通过FileLock.idshared()来判断锁的状态。

如果直接使用lock()或者tryLock()获得锁为排他锁，它适用于并发量不大的情况

**Selector**：支持非阻塞式输入输出,他是一个多路复用器，他是java NIO组件中的一个，用于检查一个或多个NIO Channel的状态是否处于可读，可写，它使用更少的线程就可以来处理通道了，相比使用较多个线程，减少的上下文切换的开销。

Selector具体是干什么的？

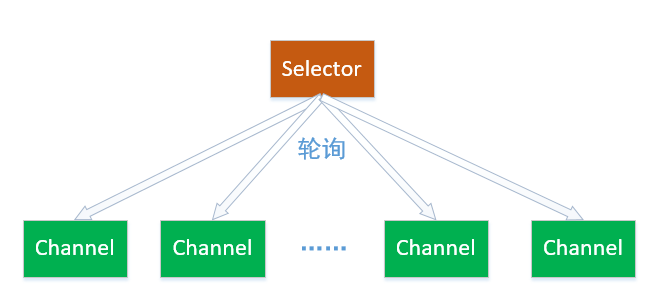
例如一个养鸡场，有这样一个人，每天的工作集会所不停的检查几个特殊的鸡笼，如果有鸡进来，有鸡出去，有鸡生蛋，有鸡生病等等，就会把相应的情况记录下来，如果鸡场的负责人想知道情况，只需要询问那个人就可以。

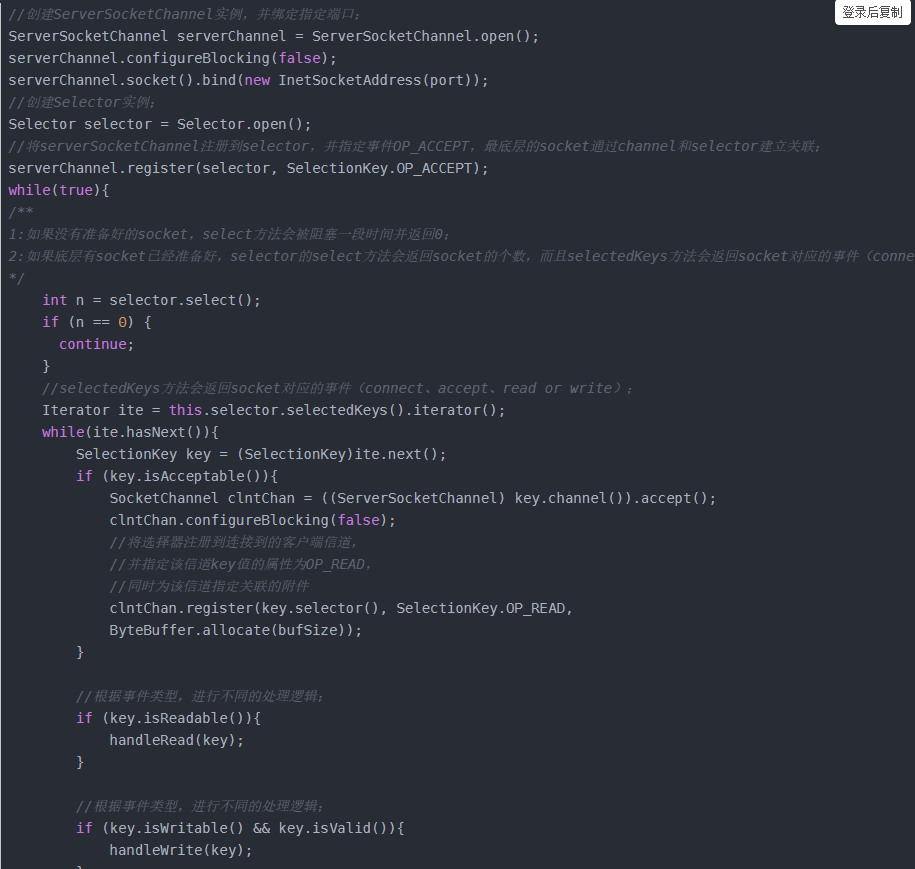
在这里这个人就是Selector，每个鸡笼相当于一个SocketChannel，每个线程通过Selector可以通过管理多个SocketChannel.

为了实现selector管理多个Socketchannel，必须将具体的SocketChannel对象注册到Selector，并声明需要监听的事件，这样Selector才能知道需要记录什么数据，监听的事件有4种：

1. Connect:客户端连接服务端事件，对应的值为SelectionKey.OP\_CONNECT(8)
2. Accept:服务端接受客户端连接事件，selection.OP\_ACCEPT(16)
3. Read：读事件，selection.OP\_READ(1)
4. Write：写事件selection.OP\_WRITE(4)

真个过程：每次请求到达服务器，都是从connect开始，connect成功后，服务端开始准备accept，准备就绪，开始读数据，最后写会数据返回。因此当SocketChannel有对应的事件发生时，selector都可以观察到，并进行相应的处理。





服务端操作过程

1. 创建ServerSocketChannel实例，并绑定指定端口；  
   2、创建Selector实例；  
   3、将serverSocketChannel注册到selector，并指定事件OP\_ACCEPT，最底层的socket通过channel和selector建立关联；  
   4、如果没有准备好的socket，select方法会被阻塞一段时间并返回0；  
   5、如果底层有socket已经准备好，selector的select方法会返回socket的个数，而且selectedKeys方法会返回socket对应的事件（connect、accept、read or write）；  
   6、根据事件类型，进行不同的处理逻辑；

**在步骤3中，selector只注册了serverSocketChannel的OP\_ACCEPT事件  
1、如果有客户端A连接服务，执行select方法时，可以通过serverSocketChannel获取客户端A的socketChannel，并在selector上注册socketChannel的OP\_READ事件。**

1. **如果客户端A发送数据，会触发read事件**

**这样下次轮询调用select方法时，就能通过socketChannel读取数据，同时在selector上注册该socketChannel的OP\_WRITE事件(逻辑在handleRead(key方法中实现))，实现服务器往客户端写数据。**

**Selector(选择器）,SelectableChannel（可选择通道）,Selectionkey（选择键）**

**Seletor选择器类管理着一个被注册的通道集合的信息和他们的就绪状态，通道是和选择器一起被注册的，并且使用选择器来更新通道的就绪状态，这么做，可以选择将被激发的线程挂起，直到有就绪的通道。**

**Selectablechannel提供了实现通道的可选择性所需要的公共方法。**

**SelecionKey：封装了特定的通道和特定的选择器的注册关系，选择键对象被SelectableChannel.register()返回并提供一个表示这种注册关系的标记。**

创建：Selector Selector=Selector.open()；

将channel注册到selector来实现Selector管理Channel,需要将Channel注册到相应的Selector上。**configureBlocking（）** 方法用于使通道处于阻塞模式或非阻塞模式。

channel.configureBlocking(false);

SelectionKey key= channel.register(selector,SelectionKey,OP\_READ);

注册后，与selector一起使用时，Channel必须处于非阻塞模式下，负责抛出阻塞异常，这意味着FileChannel与Selector一起使用，因为FileChannel不能切换到非阻塞模式，而套接字通道都可以切换到非阻塞模式下，并且通道一旦被注册，将不能再回到阻塞状态，此时若调用通道的configureBlocking(true)将抛出BlockingModeException异常，其中register()方法的第二个参数是一个兴趣集合，表示选择器所关心的通道操作，我对那块感兴趣我就只会检查通道中这个操作是否已经处于就绪状态。而且并非所有的操作在所有的可选择性通道上都能被支持，比如ServerSocketChannel支持Accept，但SocketChannel不支持，可通过validOps（）方法来获取特定通道下所支持的集合操作。如果selector对通道的多操作类型感兴趣，可以使用位或操作符实现

interestSet=SelectionKey.OP\_READ|SelectionKey.OP\_WRITE;

并且返回的SelectionKey对象（interest（是selector感兴趣的集合）,read（是通道已经就绪的操作集合）,channel,Selector（通过selectionkey访问对应的Selector和Channel，Channel channel = Selectionkey.Channel(),selector selector = Selectionkey.selector()））OPS()操作，也可通过cancel()方法取消特定的注册关系。

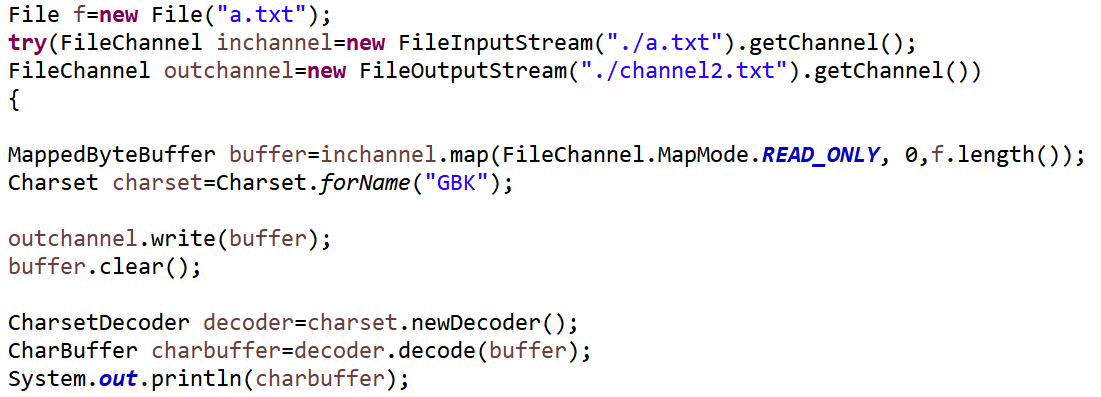
**key.attachment(); //返回SelectionKey的attachment，attachment可以在注册channel的时候指定。**

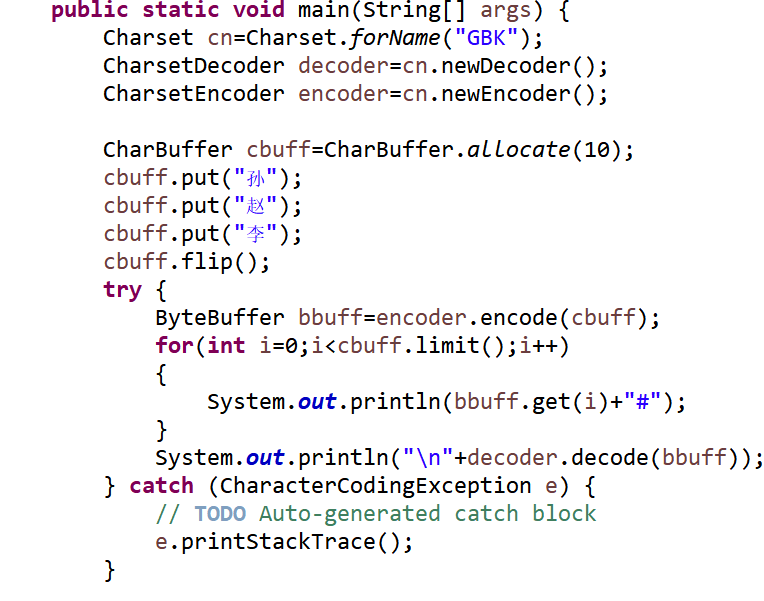
**key.channel(); // 返回该SelectionKey对应的channel。**

**key.selector(); // 返回该SelectionKey对应的Selector。**

**key.interestOps(); //返回代表需要Selector监控的IO操作的bit mask**

**key.readyOps(); // 返回一个bit mask，代表在相应channel上可以进行的IO操作。**





1. 多线程与并发（线程，线程通信，阻塞队列，线程池(Executor,ForkjionPool)，ThreadLocal，集合安全，线程与进程，syn,volicatal，concurrentHashmap实现原理，原子操作类，锁，CAS,）

在操作系统引入进程的目的是为了使得多个程序能够并发执行,提高资源利用率和吞吐量，而引入线程的目的是，因为在多个程序并发执行时，他们之间是通过轮转分配固定时间的方式来进行并发操作的，因此涉及到进程的轮流切换，而进程的上下文切换是非常消耗系统资源的一个过程，而对于线程而言：他也是可以独立运行的，当线程切换时仅需要保存和设置少量的寄存器内容（**上下文切换的过程是一个任务从保存到再加载的过程**），切换的代价较小，这样减少了并发执行所付出的时空开销，使得OS有更好的并发性，并且多线程情况下，任务的执行不一定会加快，当并发累加到上万次的时候，他和串行的的执行效率差不多，或者更慢，这是因为线程上下文切换的开销造成的，可以通过vmstat测量（cs）上下文切换的次数,那么在这种情况下如何减少线程的上下文切换？可以使用无锁并发编程，CAS算法，使用较少的线程或者使用协成的方式来减少上下文切换的次数，对于无锁并发编程可以将并发访问的数据进行分段处理，可以通过数据的ID按照hash算法取模分段，使得不同的线程访问不同段的数据，对于CAS算法：例如Atomic包中的使用CAS算法来更新数据，不需要加锁，也可以使用较少数量的线程，来缓解线程上下文切换所带来的性能影响，也可以使用协成的方式：可以在单线程中实现多任务的调度，并在单线程里维持多个任务间的切换，通过jvm调优工具可以观察一个进程里的线程在做什么事情，查看这些线程的状态，如果里面的线程有大多数是等待状态，说明线程池里接收到的线程太少，应该减少线程的数量。

进程作为系统中拥有资源的基本单位，他是程序的一次执行，包括数据段，程序段和PCB数据结构等，对于一个能够兜里运行的程序而言他是必须都对应的PCB数据结构的，而线程本身并不拥有系统资源，而是仅有一点必不可少能够使其独立运行的资源即可。

线程的创建：三种

继承Thread并重写run()方法，通过实例对象.start()进行启动，通过

实现Runnable接口，重写run()方法，通过new Tread(线程实例对象名，附加命名）.start()

使用Callable和Future创建线程，Callable接口提供另一个call()方法作为线程执行体，但Call（）方法有返回值，也可以声明报出异常，可以Callable对象作为Thread的target，因为他不是Runnable接口的子接口，所以Callable对象不能直接作为Thread的target，Call（）方法并不是直接调用，而是作为线程执行体被调用，而且Call()方法有一个返回值，这个返回值可通过Future接口代表Callable接口里call()的方法返回值，而且对应的FutureTask实现类，该实现类实现了Future接口，并实现了Runnable接口，可以作为Thread类的target

FutureTask<Integer> task = new Futuretask<Integer>((Callable<Integer>)()->{

//线程执行体

});

通过new Thread(task,”附加别名”）.start()，因为他有返回值，可以通过task.get()方法获取到。

**对于一个线程而言线程如何终止**

1. 使用退出标志，是线程正常退出，也就是线程执行体执行完后线程终止
2. 使用stop方法强制终止线程
3. 使用interrupt方法中断线程
4. 使用Future接口的cancel()视图取消Future里关联的Callable

**描述线程的声明周期**：

当新建一个线程后，启动之后就会处于就绪状态，当得到处理器则进入运行态，当调用yeild()方法或失去处理器资源后，他会从运行态转为就绪态，当运行态时调用sleep()或IO阻塞，等待同步锁，等待通知，或者线程被挂起，则会处于阻塞状态，，当上面这些情况消失后会就会又处于就绪状态，当运行态，通过线程停止的方法调用或者run(),call()执行完成后标志线程死亡。

**阻塞队列**：他是一个支持两个阻塞方法的，一个是阻塞插入方法，判满，满则阻塞，一个是阻塞移除方法，判空，空则阻塞,常用于类似生产者消费者这样的场景，对于这两种阻塞有4种处理方式，抛出异常，返回特殊值，一直阻塞，超时退出，对于阻塞队列而言，他的添加使用put(e),移除使用take（e）,他对于上述状况会一直阻塞。

Java中的7种阻塞队列：

.**ArrayBlockingQueue**:基于数组的有界阻塞队列,采用FIFO的原则对元素进行排序，默认情况不保证线程公平的访问队列，而想让其公平访问可以使用可重入锁来实现。

**.LinkedBlockQueue:**基于链表的有界阻塞队列，此队列的默认最大长度为Integer.MAX\_VALUE，按照FIFO的原则对元素进行排序。

.**SynchronousQueue**:不存储元素的阻塞队列，他的每一个put操作必须等到一个take()操作，否则不能继续添加元素，它支持公平访问队列，默认采用非公平策略进行访问，可以将他的构造参数设置为true既采用公平策略访问队列，它类似于一个中转站，负责将生产者线程处理的数据直接传递给消费者线程，队列本身并不存储元素，是一种传递性场景。

**.DelayQueue**:是一个支持延迟获取元素的无界队列，队列使用priorityQueue来实现，队列中的元素必须实现Delayed接口，在创建元素时可以指定多久才能从队列中获取当前元素，只有在延迟期满了之后才能从队列中提取元素。

使用场景，可以将DelayQueue运行在**缓存系统设计**，**定时任务调度**等场景

**在缓存系统设计上**：可以用DelayQueue保存缓存元素的有效期，使用一个线程循环查询DelayQueue，一旦能从DelayQueue中获取元素，表示缓存有效期已过。

**在定时任务调度**：使用DelayQueue保存当天将会执行的任务和执行时间，一旦从DelayQueue中获取到任务就开始执行。

**如何实现延时阻塞队列**：延时阻塞队列，当消费者从队列里获取元素时，如果元素没有达到延时时间，就阻塞当前线程。

**LinkedBlockingDeque**:由链表组成的双向阻塞队列，可以从队列的两端进行插入好人移除操作，因为他多了一个操作队列的入口，因此减少了多线程一半的竞争，在初始化时设置容量，防止队列膨胀，并且他可以运行在：工作窃取模式中。

PriorityBlockingQueue:支持优先级排序的无界阻塞队列

LinkedTransferQueue:有链表组成的无界阻塞队列

阻塞队列的实现原理：他是通过一种通知模式实现的

**Fork/jion**:因为他是ExecutorService的实现类，因此是一种比较特殊的线程池，他是一个用于并行执行任务的框架，是把一个大任务分割成若干个小任务，最终汇总每个任务结果后得到大任务结果的框架，充分利用了CPU资源，它采用一种工作窃取算法进行，因为他是通过LinkedBlockingdeque的一个双向队列来实现的，因此每次窃取都是从队列的尾口进行访问处理，工作窃取算法的优点是充分利用线程进行并行计算，减少线程间的竞争，但是在一些极端情况下还是会存在资源竞争的。

并且他在java1.8中加入了通用池的功能，这个线程池用来处理那些没有被显式提交到任何线程池的任务，通过commPool（）方法返回一个通用池，通过getCommonPoolParallelism()返回通用池的并行级别，它的运行状态一般不会改变，除非通过系统终止。

他的大体流程就是：先看该任务是否是一个具有返回值的任务，如果是可以继承ForkJionTask的一个子类，RecursiveTask，如果没有，继承他的子类RecursiveAction,两者都要实现compute，并且他们需要ForkJionpool来执行，就是创建出这个的实例，然后将继承自ForkJionTask的子类的实例对象，通过ForkJionPool的submit(task)方法进行提交，他会返回一个FutureTask类型的对象，通过这个可以获得锁需要的返回结果，然后提交之后，他会在Compute方法中进行任务分解，调用fork()方法，直到到达目标分解任务的限制时，会执行完成，在逐一进行返回，调用相应的join()方法进行合并返回，并且通过isCompletedAbnormally()方法来检查是否执行的任务已经抛出异常。

实现原理：他是通过ForkJointask数组与ForkJoinWorkerThread数组组成，forkjiontask负责将这些任务提交给forkjoinpool的任务，forkjoinworkerThread数组负责执行这些任务。

Fork()方法的实现原理，是通过pushTask（）方法异步执行的，然后返回结果，pushTask把当前任务存放在forkjionTask数组队列里，然后调用forkjoinpool的signalWork()方法唤醒或创建一个工作线程来执行任务。

Join()方法主要的作用是阻塞当前线程并等待获取结果，他会通过一个doJoin()方法得到当前任务的状态，有4种状态，已完成，被取消，信号，出现异常，通过查看，看任务是否已经执行完成，执行完则返回结果，没有则从任务数组中取出任务去执行。

**Executor**:系统启动一个新线程的成本是非常高的，它涉及与操作系统交互，需要在用户态和内核态之间相互转换，尤其是一些生存期很短的线程，而线程池在启动时既会创建大量的线程，程序将一个Runnable或Callable对象传给线程池，线程池就会执行对应的run()或者call(),当任务执行结束后，该线程不会死亡，而是再次返回线程池中成为空闲状态，然后等待执行下一个任务，这样减少了线程创建和销毁所带来的资源消耗，，并且提高了响应速度，使得线程管理更加方便，可以使用线程池进行统一分配，调优和监控。

实现原理：

首先说一下处理流程：当一个任务到达线程池之后，他会首先判断核心线程池是否已经满了，如果满了则加入到工作队列中（他是一个阻塞队列），工作队列如果满了，线程池会判断线程池中的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务，如果已经满了，则会交给**饱和策略**来处理这个任务，当上面都不满的情况，他回获取一个全局锁。

因此几乎所有的操作都是通过阻塞队列去获取任务，因为这样获取不需要加全局锁进行控制。

在主线程执行完execute()方法中创建一个线程，会让这个线程执行当前任务，当线程执行完线程池中的任务后，会循环从BlockingQueue中获取任务来执行。

他的主要参数有：

CoolPoolSize:线程池的基本大小，当提交一个任务到线程池，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需要执行的任务数大于线程池基本大小时就不再创建，可以通过prestartAllCoreThreads()方法提前创建并启动所有基本线程。

maximumPoolSize:线程池允许创建的最大数量，如果队列满了，并且已创建的线程数小于这个值则线程池会创建新线程执行任务，但如果使用无界队列，这个值就没意义了。

keepAliveTime:线程池的工作线程空闲时，保持存活的时间，如果任务很多，每个任务的执行时间比较短的，可以调大，提高线程利用率。

Milliseconds:是一个Timeunit的时间单位

runnabletaskQueue:任务队列，用于等待执行任务的阻塞队列，

有ArrayBlockingQueue（有界）,

LinkedBlockingQueue（有界阻塞队列，Executors.newFixedThreadPool()创建一个使用固定线程数的线程池）,

SynchronousQueue,（同步队列，队列本身不存储元素，对该队列的存取操作必须交替进行，其中Ececutors.newCachedThreadPool（具有缓存功能的线程池）就是基于这个实现的）

priorityBlockingQueue：具有一定的优先级。

Handler：是一个拒绝执行任务的饱和策略，可以直接抛出异常，或者丢弃队列里最近一个任务，并执行当前任务，或不处理也不执行。

向线程池中提交任务可以使用：execute()或submit()，execute没有返回值，因此一个任务执行后，不摘到它是否执行成功，而submit用于提交需要返回值的任务，线程池会返回一个future类型的对象，这个对象通过get()方法可以获取到返回值，get()方法会阻塞当前线程知道任务完成，也可以为get()方法提供阻塞时间，阻塞一段时间后立即返回。

Shutdown和shutdownnow()的区别：其原理都是遍历线程池中的线程，然后逐个调用线程的interrupt方法来中断线程，通常调用shutdown()来关闭，如果任务不一定要执行完，可以嗲用shutdownnow().

shutdown是将线程池的状态设置为SHUTDOWN状态，然后中断所有没有正在执行的任务的线程。

Shutsownnow()首先会将线程池的状态设置为STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表。

怎样配置线程池：这个是多方面的，要根据任务的性质（看他是cpu密集，IO密集，两者的混合任务），再者要根据任务的优先级，长短等来综合决定，

建议使用有界队列，因为如果系统发生异常，可能会导致任务队列和线程池都满了，他会采用饱和策略不断丢弃任务，在这种情况下如果使用无界队列，可能会使得资源消耗殆尽，导致系统崩溃。

线程池也提供了许多监控方法：taskCount()需要执行的任务数，getActiveCount()获取活动线程数。也可以通过重写线程池来达到线程池细节监控的目的。

Executor框架：它是一个两集调度模型，第一层是线程与内核态之间的调度，第二层是任务与线程之间通过Executor的调度。

它主要由三部分组成：1.任务（Runnable,Callable，他们都能通过ThreadPoolexecutor和 ScheduledthreadPoolExecutor来执行，在使用任务时可以通 过executors.Callable(runnable）将一个Runnable对象封装为一个 Callable对象）

2.任务的执行（主要通过两个核心类进行任务的执行，通常通过工厂类Executors来创建这两个的线程池ThreadPoolExecutor和ScheDduledThreadExecutor，他们实现了ExecutorService接口，它又继承自Executor接口），其中ThreadPoolexecutor用于执行被提交的任务，ScheduledThreadexecutor可以做到在给定的时间之后运行命令，或者定期执行，他比Timer更灵活，功能更强大。

3.返回结果：通过Future接口和Futuretask实现类的get()阻塞调用方法进行结果值的返回，通过cancel()来取消Callable与Future之间的绑定关系。

而对于ThreadPoolExecuto:通过Executors工厂类可以实现三种：

1. FixedThreadPool(创建固定的线程数，内部通过无界阻塞队列LinkedBlockingQueue实现，它的边界最大值为Integer.max\_value，，它的线程基本大小和线程最大容量都是传入的这个固定参数值，它在执行完后没有存活时间，因为他的keepliveTime为0，它的执行大致过程：

如果当前运行的线程数少于collPollsize的大小，则创建新线程，当线程预热完成后，会将任务加入阻塞队列中去，因为他采用无界队列进行，因此它的maxinumpoolSize和keepAliveTime将是无效参数，并且他不会拒绝任务。

,2.CachedThreadPool,

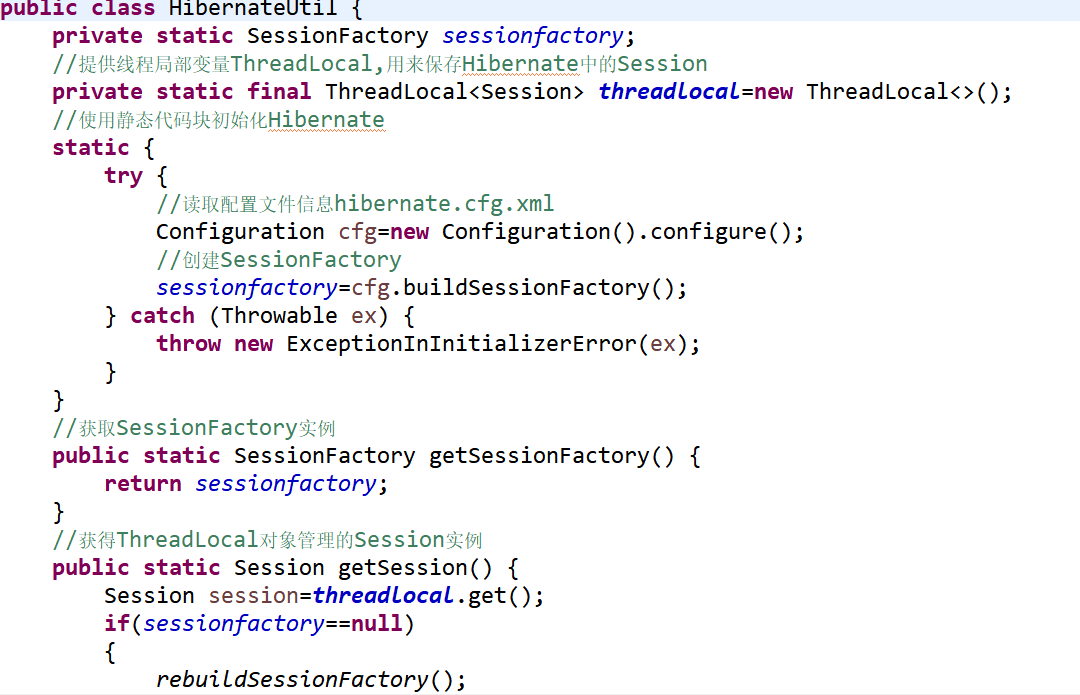
3.SingThreadexecutor

而对于ScheduledThreadPoolExecuto:通过Executors工厂类可以实现2种类型的ScheduledThreadPoolExecutor：

1. ScheduledThreadPoolExecutor:
2. SingleThreadScheduledexecutor:

而对于Executorservice.submit()所的到的返回，他是一个Futuretask类型的对象：

**ThreadLocal：**他是线程局部变量，通过他可以简化多线程编程时的并发访问，可以很方便的隔离多线程之间的资源竞争，他会为每一个使用该变量的线程都提供一个变量副本，是每一个线程都可以独立的改变自己的副本，而不会和其他副本冲突，看上去就好像是一个线程可以独立拥有该变量一样，它提供的基本增，删，概操作，他是从另一个角度去控制并发，而不是通过同步机制，会将并发访问的资源复制多份，然后每一个线程用的资源都是该并发访问资源的副本，因此就不用对资源进行同步处理了，它是通过线程隔离的方式来避免多个线程之间的资源共享，具体应用：



**Volatile关键字：1.保证了新值能立即同步到主内存，每次使用前立即从主内存刷新（既有 可见性），而普通变量的值在线程间传递均需要通过主内存来完成，它在每次使用的时候回先进行刷新，执行引擎看不到他不一致的情况，因此它是由不一致，但是不会影响数据的正常访问，但是他在并发情况下也是不安全的**

1. **它会进制指令重排序，普通情况下，所有的依赖赋值操作都是能够获取到正确结果的，但是不能保证赋值的操作顺序和程序代码的执行顺序一致。**

**在懒汉模式下可以使用双锁检测的方式，，它是并通过插入好多的内存屏障来保证程序的顺序执行，其中Lock addL 0,他是一个空操作，不用nop是因为规定不许使用，这个操作会将数据同步到主内存中。**

**Synchronized与锁：对于同步方法，锁是当前实例对象，对于静态方法，锁是当前类的class,对于同步方法，锁是Synchonized括号里匹配的方法，他是使用两个字节码指令实现的monitorenter,monitorexit.**

**可重入锁：**

**悲观锁：**他是利用数据库内部机制提供的锁的方法，对更新的数据加锁，在并发期间一旦有一个事务持有了数据数据库记录的锁，其他线程将不能再对数据进行更新了，也就是说当一条线程抢占了资源后，其他的线程将得不到资源，这个时候cpu就会将这些得不到的资源线程挂起，挂起的线程也会消耗CPU的资源，尤其是在高并发的请求中，并且使用悲观锁就会造成大量的线程被挂起和恢复，在mybatis如果通过主键进行查询，所以只对行加锁，可以在语句上加上for update语句，意味着将持有对数据库记录的行更新锁，

使用数据库的锁机制，可以消除数据不一致性，但是使用悲观锁，数据库的性能会下降，因为大量的线程被阻塞，而且需要有大量的恢复过程

**（乐观锁）CAS原理：**乐观锁是一种非阻塞锁，它不会阻塞其他线程并发的机制，因此并不会引起线程的挂起和恢复，这样能够提高并发的能力。

CAS原理并不排斥并发，也不独占资源，只是在线程开始阶段就读入线程共享数据，保存为旧值，当处理完逻辑，需要更新数据的时候，会进行一次比较，既比较线程共享的数据是否和旧值保持一致，如果一致，就更新数据，如果不一致，说明该线程已经被其他线程修改了，然后可以进行重试，并且他会出现ABA问题，通过版本号加以控制，可避免此类问题的发生，但这样的话在数据量比较大的情况下回因为版本不一致的而造成大量的请求失败的情况，因此这块可以使用可重入机制，这样可能会造成大量的sql重新执行，因此可以通过设置时间，或者限制重入次数来提高请求的成功率。

使用乐观锁有助于提高并发性能，但是由于版本冲突，乐观锁导致多次请求服务失败的概率大大提高，通过乐观锁的重入机制，按照时间戳或者按次数限定来提高成功的概率，但是使用这种方式会随着版本号的冲突的概率提升而提升，并且使用乐观锁会导致SQL大量执行，对数据库的性能要求较高，容易引起数据库性能的瓶颈，而且重入机制的开发比较复杂，对于项目的开发的进度也是一个巨大的挑战。

**Redis：redis可以去实现高并发，它类似于一个缓存器的功能，并且通过redis提供的lua脚本的原子性，消除了数据的不一致性问题，它有6种数据类型，String，List（链表），set(无序），Hash(哈希散列表），Zset(有序集合），HyperLogLog。**

**它的数据结构有：字符串，哈希，链表，集合，有序集合，基数。**

**有序集合：它是在set集合的集合的基础之上，加了一个分数，这个分数是一个浮点数，根据这个分数，redis就可以支持对分数从小到大或者从大到小排序，并且他对于每一个元素而言是唯一的，对于不同的元素，他的分数可以一样，因为他是set，其实他是通过hash表实现的，所以他的添加，删除，等操作都是O（1）的复杂度。**

**Redis的事务，multi,watch key,exec,discard,他在这块是参考了CAS的实现过程，**

**发布订阅模型：subscribe chat**

**Publish chat “ ”是一种消息通信通信模式，他是通过继承于Messagelistener，它是通过饭序列换的方式实现的。**

**原子操作类：**

**1.原子更新基本类型：**Atomicinteger，get()返回当前值，addAndGet(int delta)，compareAndSet(int expect,int update),int getAndincrement(),,void lazySet(int new value),int getAndSet(int newvalue).当他调用原子自增操作后，会在getAndincrement()方法中进行CAS操作，在调用了compareAndSet方法后，他会调用unsafe的compareAndSwapInt()方法进行更新操作，如果操作成功，返回自增前的值，如果操作失败，则会重新进行CAS操作。而unsafe提供了3种基本类型的更新操作，compareAndSwapObject(object 0,long offset,Object expected,object x),compareAndswapInt(),compareAndswapLong()的本地方法

**2.原子更新数组：**AtomicIntegerArray(提供常用方法有addAndGet(int i,int delta)方法和compareAndSet(int i,int expect,int update)方法等），AtomicReferenceArray

1. **原子更新引用类型：AtomicReference**
2. **原子更新字段类:AtomicIntegerFieldUpdater,AtomicStampedReference(原子更新带有版本号的引用类型，解决了ABA问题）**
3. Jvm
4. 数据结构与算法
5. Ssm,ssh，Shiro原理机制分析
6. Mysql,NOSQL,数据库索引
7. 设计模式
8. 计算机网络
9. 编译器
10. 汇编语言
11. Negix和tomcat
12. git和svn
13. 分布式
14. Linix常用命令
15. 面经